VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Porovnání vlastností různých implementací uzlů datových struktur

Comparison of Features of Different Data Structure Implementations

2015 David Holuša

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. David Holuša**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Porovnání vlastností různých implementací uzlů datových struktur
Comparison of Features of Different Data Structure Implementations

Zásady pro vypracování:

Práce bude porovnávat vlastnosti různých implementací uzlu datových struktur a to z pohledu vkládání, aktualizace a mazaní. Práce vyjde z existující implementace vyvíjené databázovou skupinou na Katedře informatiky, VŠB-TU Ostrava. Práce bude probíhat v následujících krocích:

1. Rozšíření stávající implementace B-stromu a sekvenčního pole o možnost mazaní a aktualizace prvků.
2. Přidání zápisu prováděných operací do log souboru a implementace zotavení z chyby.
3. Příprava benchmarku.
4. Porovnání výkonu benchamrku při zapnutém logování a bez něj pro všechny operace a pro obě datové struktury (B-strom a sekveční pole).

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 6. května 2015 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2015 .....

Tímto bych chtěl poděkovat vedoucímu mé diplomové práce Ing. Radimu Bačovi, Ph.D. za odborné vedení této práce, za čas strávený konzultacemi a za poskytnutí mnoha cenných rad a informací. Především děkuji za velkou míru ochoty a trpělivosti, díky kterým mohla tato práce vzniknout.

Abstrakt

Diplomová práce se zabývá problematikou stránkovaných datových struktur a transakčním zpracováním. Jsou zde popsány datové struktury B-strom, B+strom a sekvenční pole. Transakční zpracování je zde pojato v souvislosti se zotavením transakce a systému. S tím souvisí vymezení pojmu transakce a seznámení se základními technikami zotavení.

V další části této práce je popsán databázový rámec RadegastDB, který je vyvíjen databázovou skupinou na katedře informatiky VŠB.

V praktické části je pak popsána implementace rozšíření rámce RadegastDB o podporu transakčního zpracování a možnost mazání a aktualizace v B+stromu.

Závěrečná část je věnována měření vlivu transakčního zpracování na výkon prováděných operací v B+stromu a sekvenčním poli.

Klíčová slova: Databáze, B-strom, B+strom, Sekvenční pole, Datové struktury, Transakce, Zotavení transakce, RadegastDB

Abstract

This thesis deals with problems of paged data structures and transaction processing. There are described data structures B-tree, B+tree and sequential aray. Transaction processing is conceived in relation to the transaction and system recovery. This is related to definition of transaction and introduction to the basic techniques of recovery.

In next part of this work is described RadegastDB database framework which is developing of database group at the Department of Computer Science in VŠB.

In the practical part is described imlementation of RadegastDB framework extended to support of transaction processing and the possibility of deleting and updating the B+tree.

The final section is devoted to measurement of the impact of transaction processing on performance of operations at the B+tree and sequential array.

Keywords: Database, B-tree, B+tree, Sequential Array, Data Structure, Transaction, Transaction Recovery, RadegastDB

Seznam použitých zkratek a symbolů

DS	–	Datová struktura
IT	–	Informační technologie
SŘBD	–	Systém Řízení Báze Dat

Obsah

Seznam tabulek

Seznam obrázků

List of Algorithms

1 Úvod

1.1 Motivace

Stejně jako v běžném životě, tak i ve světě IT je častým problémem efektivní a dostatečně výkonná práce s daty.

V běžném životě to mohou být nejrůznější tištěné dokumenty (smlouvy, lékařské záznamy, knihy, atd.), které potřebujeme nějakým efektivním způsobem uložit, vyhledávat, aktualizovat nebo odstranit. K tomu nám poslouží systém v podobě šuplíků, kartoték, pořadačů nebo jiných nástrojů ke kategorizaci dokumentů (dat).

Ve světě IT máme obdobný problém. Potřebujeme ukládat velké množství dat nejrůznějšího charakteru (obrázky, dokumenty, knihy, atd.). K tomu nám slouží databáze. Pojem databáze může být chápán jako uspořádaná množina informací. V širším slova smyslu se ale jedná o systém, který nám umožňuje přístup a práci s daty. Jde o tzv. Systém Řízení Báze Dat (SŘBD). Pokud se vrátím k příkladu se šuplíky s prvního odstavce, tak SŘBD můžeme chápat jako kvalifikovaného pracovníka, který jednotlivé dokumenty ukládá do daných šuplíků, vyhledává, aktualizuje a odstraňuje. Otázkou ale zůstává, co je v pojetí databází chápáno jako šuplík? Jak již bylo zmíněno, tak databáze je mimo jiné chápána jako množina informací, která je uspořádaná do, tzv. datové struktury. Datové struktury rozdělujeme podle uspořádání jednotlivých uzlů (stromové, sekvenční, atd.). Tyto uzly pak můžeme chápat, jako šuplík s našeho příkladu.

Každá datová struktura má své výhody i nevýhody. Proto určitě stojí za to porovnat vlastnosti jednotlivých datových struktur abychom byli schopni vybrat pro naše řešení tu nejvhodnější. Toto je také jeden s důvodů vzniku této diplomové práce. [4]

1.2 Cíle

Výstupem této práce by mělo být splnění následujících bodů:

- Rozšíření existující implementace rámce RadegastDB, který je již několik let vyvíjen databázovou skupinou na katedře informatiky VŠB. Rámec by měl být rozšířen o možnost mazání, aktualizace a zápis do log souboru pro datové struktury B+strom a sekvenční pole.
- Porovnání výkonu obou datových struktur pro všechny operace (vkládání, mazání a aktualizace) při zapnutém, resp. vypnutém zápisu do log souboru.

1.3 Struktura textu

Text je členěn do 8 kapitol ve kterých bude čtenář seznámen s obecnou teorií datových struktur a transakcí, přes úvod do rámce RadegastDB, implementaci, výsledky experimentů až k celkovému zhodnocení práce. Obsah jednotlivých kapitol je následující:

Kapitola 2. uvádí do problematiky stránkovaných datových struktur a seznamuje čtenáře s B+stromem a sekvenčním polem.

Kapitola 3. popisuje využití transakcí a uvádí do problematiky zotavení a technik zotavení.

Kapitola 4. seznamuje čtenáře s rámcem RadegastDB, jeho historií a strukturou.

Kapitola 5. popisuje první část implementace rozšíření rámce RadegastDB o možnost mazání aktualizace. Dále popisuje problémy, které při implementaci nastaly.

Kapitola 6. popisuje druhou část implementace rozšíření rámce RadegastDB o transakční zpracování.

Kapitola 7. obsahuje vyhodnocení všech experimentů pro řešení z předchozí kapitoly.

Kapitola 8. uzavírá celou práci a shrnuje její výsledky a přínosy.

2 Stránkované datové struktury

Předpokladem pro uložení dat do datové struktury je schopnost perzistence, tedy uložení do nějaké vnější paměti. Důvodem je především umožnění přístupu k datům i po neočekávaném selhání systému, vypnutí počítače, apod. Datové struktury jsou obecně **stránkované**, tzn. jsou tvořeny různým počtem stánek. Pokud si např. vezmeme libovolnou stromovou DS, pak jednu stránku reprezentuje uzel stromu. Každá stránka DS je pak mapována do stránek vnější paměti. Pokud je vnější pamětí např. pevný disk, pak základní jednotkou je tzv. diskový blok (nejčastěji o velikost 512B). Tyto diskové bloky jsou pak organizovány od alokačních jednotek (nejčastěji o velikosti 2048 nebo 4096B) se kterými pracuje souborový systém. Snahou tedy je, aby stránky DS měli velikost, která násobkem diskového bloku. Důvodem je dosáhnout pokud možno, co nejmenšího počtu I/O operací [4].

2.1 B-strom

Jedním z příkladů stránkované datové struktury je B-strom, kterou v roce 1970 navrhl R.Bayer [1]. Stránky jsou zde reprezentovány jednotlivými uzly B-stromu. Jde o velice efektivní datovou strukturu, která má v nejhorším případě složitost vyhledávání prvku $log_n(N)$, kde N je počet prvků ve stromu. Další významnou vlastností B-stromu je faktor využití paměti, který je minimálně 50%. [2]

Definice 2.1 *B-strom řádu n je (2n+1)-ární strom, který splňuje následující kriteria [2]:*

1. *Každá stránka obsahuje nejvýše 2n položek (klíčů).*
2. *Každá stránka, s výjimkou kořenové obsahuje alespoň n položek.*
3. *Každá stránka je buď listovou tj. nemá žádné následovníky nebo má m+1 následovníků, kde m je počet klíčů ve stránce.*
4. *Všechny listové stránky jsou na stejné úrovni.*

2.2 B+strom

Další stránkovanou datovou strukturou je B+stromu, který vychází z B-stromu. Oproti B-stromu má B+stromu uložené klíče pouze v listových uzlech. Klíče ve vnitřních uzlech jsou pouze rozhodovací. Dalším specifikem B+stromu je provázanost listových uzlů, tzn. že každý listový uzel má ukazatel na svého levého i pravého souseda. Díky této vlastnosti můžeme položky B+stromu procházet sekvenčně bez nutnosti procházet celým stromem. [5][3]

Definice 2.2 *B+strom řádu n je n-ární strom, který splňuje následující kriteria [5]:*

1. *Kořen má nejméně dva potomky, pokud není listem.*
2. *Každý uzel kromě kořene a listu má nejméně n/2 a nejvýše n potomků.*

3. *Každý uzel má nejméně n/2-1 a nejvíce n-1 položek (klíčů).*

4. *Všechny větve od kořene do listu jsou stejně dlouhé.*

5. *Položky v nelistovém uzlu jsou organizovány následovně:*

$$p_0, (k_1, p_1), (k_2, p_2), ..., (k_n, p_n), u$$

$p_0, p_1, ..., p_n$ jsou ukazatele na potomky,
$k_0, k_1, ..., k_n$ jsou klíče,
u je nevyužitý prostor,
(k_i, p_i) jsou záznamy uspořádány vzestupně podle klíčů, přičemž $n/2 - 1 \leq m \leq n - 1$

6. *Odpovídá-li ukazateli p_i, kde $i \in < 1, n >$, podstrom $U(p_i)$, potom platí:*

   (a) pro každé k v $U(p_{i-1})$ je $k \leq k_i$
   (b) pro každé k v $U(p_i)$ je $k > k_i$

7. *Listy obsahují úplnou množinu klíčů.*

2.2.1 Vkládání

Na jednoduchém příkladu 2.2 si demonstrujeme princip vkládání a s tím související štěpení a růst stromu.

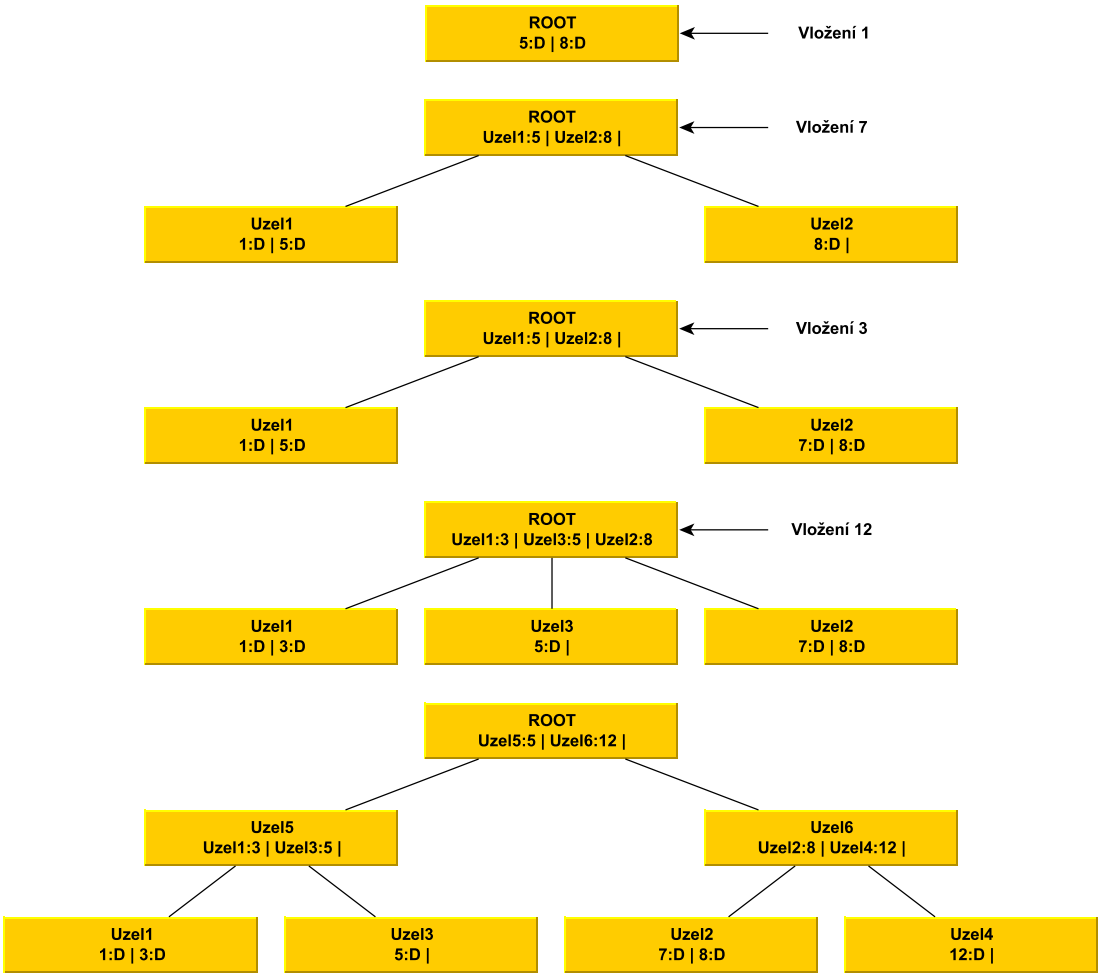
Příklad 2.1
Předpokládejme, že máme strom, kde minimální počet prvků v uzlu je 1 a maximální počet prvků je pro listový uzel 2 a vnitřní uzel 3. Prvek je ve vnitřním uzlu chápán jako dvojice ukazatel na potomka a klíč, proto je maximální počet prvků o jedeno větší než u listového.

Nyní budeme postupně vkládat sekvenci klíčů 5, 8, 1, 7, 3, 12. Průběh vkládání je patrný na obrázku 1. Strom vyobrazený na obrázku má prvky ve vnitřním uzlu uspořádány jako *ukazatel:klíč* oddělené znakem "|". V listovém uzlu jsou prvky uspořádány jako *klíč:data* oddělené znakem "|". Data jsou v tomto případě reprezentovány znakem "D".

1. Klíče 5 a 8 vložíme přímo do kořenového uzlu "ROOT", který je na začátku zároveň listovým uzlem.

2. Klíč 1 již nemůžeme do kořenového uzlu "ROOT"vložit, protože obsahuje maximální počet prvků. Proto je potřeba provést štěpení uzlu "ROOT". Při štěpení dojde k vytvoření uzlu "Uzel2"a z kořenového uzlu "ROOT"se stane uzel "Uzel1". Dále je vytvořen nový kořenový uzel "ROOT"s ukazateli na oba potomky, čímž dojde k růstu výšky stromu. Nyní již můžeme klíč 1 vložit do uzlu "Uzel1".

3. Klíč 7 je možné vložit do uzlu "Uzel2".



Obrázek 1: Vkládání do B+stromu

4. Klíč 3 je potřeba vložit do uzlu "Uzel1", kde však není místo. Proto je potřeba provést štěpení uzlu "Uzel1", čímž vznikne nový listový uzel "Uzel3"s klíčem 5. V uzlu "Uzel1", tak vznikne prostor pro vložení klíče 3.

5. Klíč 12 je potřeba vložit do uzlu "Uzel2", kde však není místo. Proto je potřeba provést štěpení uzlu "Uzel2", čímž vznikne nový listový uzel "Uzel4"s klíčem 12. V uzlu "Uzel2", tak vznikne prostor pro vložení klíče 12. Štěpení je však propagováno až do kořenového uzlu "ROOT", kde však již není dostatek místa pro další prvek. Proto je potřeba provést také štěpení kořenového uzlu "ROOT". Z uzlu "ROOT"se tak stane uzel "Uzel5"a je vytvořen nový uzel "Uzel6". Dále je vytvořen nový kořenový uzel "ROOT", čímž dojde k růstu výšky stromu.

■

2.2.2 Mazání

Na jednoduchém příkladu si demonstrujeme princip mazání a s tím související slučování a přenos prvků.

Příklad 2.2
Předpokládejme, že máme strom, kde minimální počet prvků v uzlu je 1 a maximální počet prvků je pro listový uzel 2 a vnitřní uzel 3. Prvek je ve vnitřním uzlu chápán jako dvojice ukazatel na potomka a klíč, proto je maximální počet prvků o jedeno větší než u listového.

Nyní budeme postupně mazat sekvenci klíčů 12 a 9. Průběh mazání je patrný na obrázku 2. Strom vyobrazený na obrázku má prvky ve vnitřním uzlu uspořádány jako *ukazatel:klíč* oddělené znakem "|". V listovém uzlu jsou prvky uspořádány jako *klíč:data* oddělené znakem "|". Data jsou v tomto případě reprezentovány znakem "D".

1. Klíč 12 je smazán z uzlu "Uzel5". Po smazání však nemá uzel "Uzel5"dostatečný počet prvků a proto je potřeba provést sloučení se sousedním uzlem nebo přenos prvků ze sousedního uzlu. Z uzlu "Uzel4"je proto přenesen klíč 9. Změna klíče v uzlech "Uzel4"a "Uzel5"je propagována dále na předka až ke kořenového uzlu "ROOT". V uzlu "Uzel7"je změněn prvek Uzel4:9 na Uzel4:8 a prvek Uzel5:12 na Uzel5:9. V kořenovém uzlu "ROOT"je změněn prvek Uzel7:12 na Uzel7:9.

2. Klíč 9 je smazán z uzlu "Uzel5". Po smazání však nemá uzel "Uzel5"dostatečný počet prvků a proto je potřeba provést sloučení se sousedním uzlem nebo přenos prvků ze sousedního uzlu. Sousední uzel "Uzel4"nemá dostatečný počet prvků pro přenos a proto je provedeno sloučení uzlu "Uzel4"a uzlu "Uzel5". Uzel "Uzel5"zanikne a zůstane uzel "Uzel4". Sloučení je propagováno dále na předka. Uzel "Uzel7"nemá po sloučení dostatečný počet prvků a proto je potřeba provést sloučení na úrovní vnitřních uzlů nebo přenos prvků na úrovni vnitřních uzlů. Z uzlu "Uzel6"je proto do uzlu "Uzel7"přenesen prvek Uzel3:7. Tím získá uzel "Uzel7"nového potomka, resp. uzel "Uzel7"získá nového předka. Zde můžeme vidět využití provázanosti listových uzlů, která je pro B+strom typická. Změny klíčů v uzlu "Uzel6"a "Uzel7"jsou

Obrázek 2: Mazání z B+stromu

dále propagovány ke kořenovému uzlu "ROOT". V kořenovém uzlu "ROOT"je změněn prvek Uzel6:7 na Uzel6:6 a prvek Uzel7:9 na Uzel7:8.

■

2.3 Sekvenční pole

Sekvenční pole je podle Piotra Wróblewského [6] datová struktura, která je paměťově velice úsporná a umožňuje nám uspořádání libovolného počtu prvků. Počet prvků je omezen pouze velikostí dostupné paměti.

Oproti stromové datové struktuře se jedná se o lineární stránkovanou datovou strukturu, která je o poznání méně efektivnější něž zmíněný B-strom, resp. B+strom. Složitost vyhledávání je zde lineární v závislosti na počtu prvků. [2]

Sekvenční pole je udržováno setříděné pomocí, tzv. **oblasti přetečení**, což je oblast na disku, kde jsou ukládány pouze ukazatele na nové záznamy. V oblasti přetečení jsou pak jednotlivé záznamy setříděné. V sekvenčním poli jsou jednotlivé záznamy nesetříděné z důvodu rychlosti ukládání. Je zřejmé, že třídění celého sekvenčního pole při každém vkládání by bylo časové náročné. Tento přístup má však i svá úskalí. Jednotlivé záznamy jsou časem fragmentovány mezi více stránek, což komplikuje vyhledávání a zvyšuje přístupy na disk. Pokud je fragmentace a rychlost vyhledávání neúnosná, pak je provedena tzv. reorganizace sekvenčního pole, kdy dojde k jeho setřídění. [5]

3 Transakce a zotavení z chyb

V této kapitole si řekneme něco o transakcích a to v souvislosti se zotavením systému. Je ale potřeba uvést, že transakce souvisí i s řešením problému víceuživatelského přístupu k SŘBD. Jelikož víceuživatelský přístup není předmětem této diplomové práce, tak se touto problematikou nebudeme zabývat.

3.1 Transakce

Definice 3.1 *Transakce je logická (nedělitelná, atomická) jednotka práce s databází, která začíná operací BEGIN TRANSACTION a končí operacemi COMMIT nebo ROLLBACK.[4]*

Transakce obecně nezahrnuje pouze jednu operaci, ale obvykle posloupnost operací. Každá transakce musí splňovat následující **vlastnosti (ACID)** [4]:

- **Atomicity** (atomičnost) - Transakce musí být atomická, tzn. nedělitelná.
- **Correctness** (korektnost) - Transakce musí zajišťovat převod jednoho korektního stavu do druhého.
- **Isolation** (izolovanost) - Změny provedené transakcí jsou pro ostatní transakce viditelné až po potvrzení změn (COMMIT).
- **Durability** (trvalost) - Po potvrzení změn (COMMIT) musí být zajištěna trvalost, tzn. změny jsou dostupné i po pádu systému.

V SŘBD jsou transakce řízeny pomocí komponenty manager transakcí (angl. **Transaction Management**), který implementuje operace [4]:

- **BEGIN TRANSACTION** - Znamená začátek transakce.
- **COMMIT** - Znamená úspěšné dokončení transakce. Uloží všechny změny, které byly během transakce provedeny.
- **ROLLBACK** - Znamená neúspěšné provedení transakce. Zruší všechny změny, které byly během transakce provedeny.

3.2 Zotavení transakce

Abychom byli schopni provést operaci ROLLBACK je potřeba v průběhu celé transakce zaznamenávat všechny změny do tzv. **log souboru**. Log soubor je podle J. Zendulky *"Posloupnost záznamů žurnálu zaznamenávající všechny modifikace databáze"*. [7]

Rozlišujeme následující typy záznamů žurnálu [3]:

1. $< start, T_i >$, udává že transakce T_i byla spuštěna.
2. $< write, T_i, X_i, H_1, H_2 >$, udává že transakce T_i změnila prvek databáze X_i z hodnoty H_1 na hodnotu H_2.

3. $< read, T_i, X_i >$, udává že transakce T_i měla načtenou hodnotu prvku X_i.

4. $< commit, T_i >$, udává že transakce T_i byla úspěšně dokončená.

5. $< abort, T_i >$, udává že transakce T_i byla zrušena.

V SŘBD se o zápis do log souboru stará komponenta **manager log souboru a zotavení** (angl. **Log Management and Recovery**).

SŘBD obecně nepracuje přímo s datovým souborem, ale využívá tzv. **vyrovnávací paměť** umístěnou v hlavní paměti. Důvodem je především efektivnější práce se stránkami DS. To však sebou nese určitá úskalí. Může se např. stát, že ve vyrovnávací paměti budou potvrzené změny, které ještě nebyly zapsány do datového souboru. Systém musí umožnit zotavení i v tomto případě. Databáze pak musí obsahovat všechny potvrzené změny. Toho je dosaženo, tzv. **pravidlem dopředného zápisu** do log souboru. V principu se jedná o zápis do log souboru ještě před ukončením operace, v tomto případě operace COMMIT. [4]

3.3 Zotavení systému

V předchozí kapitole jsme si popsali zotavení, tzv. **lokálních chyb** (přerušení transakce uživatelem, syntaktická chyba SQL dotazu, atd.). Kromě lokálních chyb však rozlišujeme **chyby globální**, které mají vliv na všechny transakce systému. [4]

- Systémová chyba - někdy také soft crash (např. výpadek proudu).
- Chyba média - někdy také hard crash (zničení části databáze).

3.3.1 Zotavení po systémové chybě

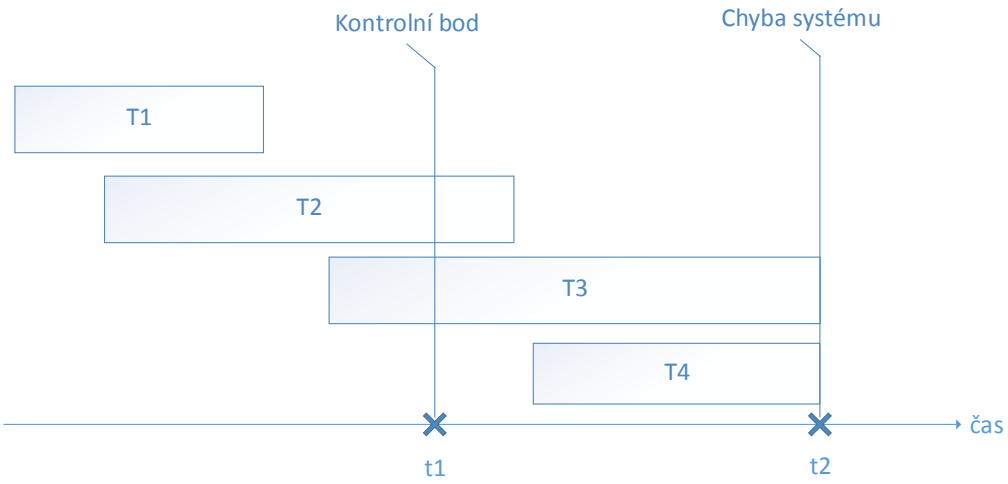
Jelikož při systémové chybě dochází ke ztrátě obsahu vyrovnávací paměti, tak neznáme přesný stav transakce, která byla přerušena chybou. Hlavním cílem zotavení je proto transakci zrušit. Zrušení transakce také nazýváme **UNDO**. Pokud je však přerušená transakce úspěšně dokončená, ale není zapsána do datového souboru, pak je nutné tuto transakci přepracovat. Přepracování pak nazýváme **REDO**. Aby byl systém schopný rozlišit, které transakce je potřeba přepracovat a které zrušit, jsou zavedeny tzv. **kontrolní body** (angl. **check point**). Záznam o kontrolním bodu je pak zapsán do log souboru. [4]

Podle J. Zendulky je kontrolní bod *periodické ukládání vyrovnávacích pamětí žurnálu a databáze na disk z důvodu snížení režie související se zotavením po výpadku*. [7]

Princip fungování kontrolních bodů a zotavení po systémové chybě si demonstrujeme na příkladu 3.1.

Příklad 3.1
Mějme transakce T1, T2, T3 a T4 z obrázku 3. Předpokládejme, že v čase t1 dojde k zápisu kontrolního bodu a v čase t2 nastane systémová chyba. Pak

- Transakce T1 byla úspěšně dokončena ještě před kontrolním bodem v čase t1 a proto není potřeba provádět žádnou akci.



Obrázek 3: Čtyři typy transakcí vzniklých při systémové chybě

- Transakce T2 byla úspěšně dokončena po kontrolním bodu v čase t1, ale nebyla zapsána do datového souboru. Proto je potřeba tuto transakci přepracovat, tedy aplikovat akci REDO.
- Transakce T3 byla spuštěna ještě před kontrolním bodem v čase t1, ale nebyla dokončena před chybou systému v čase t2. Proto je potřeba tuto transakci zrušit, tedy aplikovat akci UNDO.
- Transakce T4 byla spuštěna po kontrolním bodě v čase t1 a nebyla dokončena před chybou systému v čase t2. Proto je potřeba tuto transakci zrušit, tedy aplikovat akci UNDO.

■

3.3.2 Zotavení po chybě média

Jelikož při chybě média dochází ke ztrátě databáze nebo její části, tak samotný log soubor nám k zotavení stačit nebude. Abychom byli schopní provést zotavení po chybě média, budeme potřebovat zálohu (kopii) databáze. Zotavení je pak prováděno ve dvou krocích: [4][7]

1. Obnovení databáze ze zálohy.
2. Přepracování transakcí uložených v log souboru (použita akce REDO).

3.4 Základní techniky zotavení

V této kapitole si ukážeme několik používaných technik zotavení. V praxi jsou pak tyto techniky kombinovány s technikou kontrolních bodů.

3.4.1 Zotaveni odloženou aktualizací

Tato technika spočívá v zápisu změn provedených během transakce až po dosažení potvrzovacího bodu. V průběhu transakce jsou změny uloženy pouze v lokální vyrovnávací paměti. Po dosažení potvrzovacího bodu jsou změny zapsány do log souboru a uloženy na disk. Poté je proveden zápis změn do databáze. Pokud během transakce nastane chyba, je patrné, že není potřeba provádět operaci UNDO, protože ještě doposud nebyla transakce zapsána fyzicky do databáze. V případě chyby tedy stačí provést pouze operaci REDO a to jenom v případě, pokud byly změny zapsány do log souboru, ale nebyly zapsány do databáze. Díky tomuto pravidlu se tato technika nazývá také **NO-UNDO/REDO** algoritmus.

Je zřejmé, že algoritmus počítá s pravidlem dopředného zápisu do log souboru. V principu nám do log souboru stačí zapisovat pouze nové hodnoty, abychom byli schopni provést operaci REDO. [3]

Pokud se vrátíme k našemu příkladu 3.1 z kapitoly 3.3.1, pak transakce T3 a T4 budou oproti předchozímu řešení ignorovány. Důvodem je chybějící záznam o potvrzovacím bodu v log souboru. Transakce T1, T2 a T3 jsou řešeny podle předchozího řešení.

3.4.2 Zotavení okamžitou aktualizací

Tato technika spočívá v zápisu změn provedených během transakce ještě před dosažením potvrzovacího bodu. Již v průběhu transakce jsou změny zapsány do log souboru. Poté je proveden zápis změn do databáze. Pokud během transakce nastane chyba a transakce doposud nedosáhla potvrzovacího bodu, pak je patrné, že bude potřeba provést operaci UNDO. Pokud během transakce nastane chyba a transakce již dosáhla potvrzovacího bodu, pak je provedena operace REDO. Díky tomuto principu se tato technika nazývá také **UNDO/REDO** algoritmus. Pokud jsou všechny transakce zapsané do databáze nepotvrzené, pak je zřejmé, že není potřeba provádět operaci REDO. V takovém případě mluvíme o této technice jako o **UNDO/NO-REDO** algoritmu.

Je zřejmé, že i v tomto případě algoritmus počítá s pravidlem dopředného zápisu do log souboru. Abychom byli schopni provést operaci UNDO, je potřeba do log souboru zapisovat jak nové, tak původní hodnoty. [3]

4 Rámec RadegastDB

RadegastDB je databázový rámec, který je již několik let vyvíjen databázovou skupinou na katedře informatiky VŠB. Tento rámec je napsán v programovacím jazyce C++ a umožňuje testovat různé perzistentní datové struktury. V době psaní této práce jsou podporovány následující datové struktury:

- B+strom
- R-strom
- Q-strom
- Sekvenční pole
- Hašovací pole

V této kapitole se seznámíme s vybranými třídami, ukážeme si strukturu celého rámce a stručně popíšeme implementaci B+stromu a sekvenčního pole. Právě tyto dvě DS jsou rozšířeny v implementační části této diplomové práce, která je popsána v následující kapitole 5.

Pro lepší pochopení implementační části si také vysvětlíme používanou terminologií a některé pojmy se kterými se můžeme v tomto rámci setkat.

4.1 Základní struktura rámce

Za základ celého rámce můžeme považovat třídu cNodeCache a cMemoryPool. Obecně nám tyto třídy pomáhají vytvářet perzistentní datové struktury. Další důležitou součástí je třída cQuickDB (viz obrázek 4).

cQuickDB

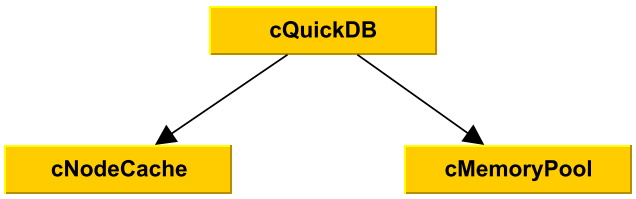
Třída cQuickDB vytváří a uchovává instance cNodeCache a cMemoryPool. Samotná instance cQuickDB je následně předávána při vytváření konkrétní datové struktury.

cNodeCache

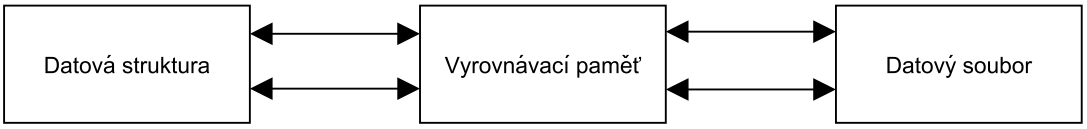
Důležitou součástí celého rámce je třída cNodeCache, která zajišťuje práci s datovým souborem. Tato třída je tady jakousi mezivrstvou samotné DS a fyzickým datovým souborem na disku.

V principu jsou uzly datové struktury načítány do operační paměti, což umožňuje rychlejší práci s těmito uzly. Veškeré operace nad DS (vkládání, čtení, mazání, atd.) jsou tedy prováděny v rámci operační paměti. Zápis do datového souboru je prováděn až v případě potřeby (explicitní požadavek na uložení DS, ukončení transakce[1], atd.). Schéma použití je možné vidět na obrázku 5.

[1]Transakce v původní implementaci podporovány nejsou, avšak jsou předmětem implementační části této diplomové práce. Implementované operace COMMIT a ROLLBACK využívají právě metody pro zápis do datového souboru (viz kapitola 6)



Obrázek 4: Třída cQuickDB



Obrázek 5: Schéma fungování vyrovnávací paměti

Třída cNodeCache navíc zajišťuje zamykání načtených uzlů, což zajišťuje konzistenci celé datové struktury. Zámky jsou dvojího druhu, jeden je pro čtení a druhý pro zápis. Každý načtený uzel je potřeba vždy odemknout.

Vyrovnávací paměť je sdílená skrz celý rámec.

cMemoryPool

Jedná se o pomocnou paměť využívanou při vykonávání operací nad DS. Stejně jako cNodeCache je i cMemoryPool sdílený skrz celý rámec.

4.2 Stránkovaná datová struktura

Všechny stránkované DS, které nám rámec nabízí, jsou v základu reprezentovány strukturou tříd, kterou můžeme vidět na obrázku 6.

cDStructHeader

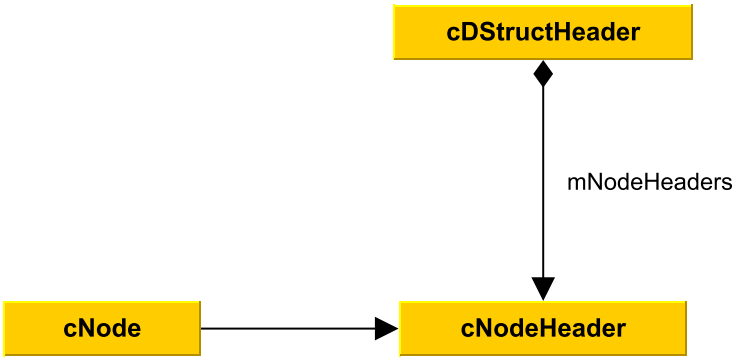
Uchovává veškeré informace týkající se celé DS (např. pole hlaviček uzlů, statistiky, unikátní jméno DS, index kořenového uzlu, atd.).

cNodeHeader

Uchovává informace týkající se jednoho typu uzlu (v našem případě listový nebo vnitřní). U stromových struktur obsahuje klíčové informace o rozložení dat v uzlu.

cNode

Reprezentuje jeden uzel DS. Je důležité zmínit, že kromě běžných lokálních proměnných obsahuje především pole mData, kde je uložen veškerý obsah uzlu. Data jsou takto uložena z důvodu výkonnosti.



Obrázek 6: Schéma obecné stránkované DS

4.3 Stromová datová struktura

Všechny stromové DS využívají základní strukturu tříd pro tvorbu stránkovaných DS. Navíc je struktura tříd rozšířena o další třídy, jak můžeme vidět na obrázku 7.

cPagedTree

Tvoří jádro každé stromové DS. Nabízí nám metody pro vytvoření a otevření DS a dále pro načítání uzlů z cache. Jelikož se jedná o jakési zobecnění, tak tato třída přímo neimplementuje operace pro vkládání, mazání a aktualizaci. Tyto operace jsou následně implementovány v odvozených třídách (např. pro B+strom, R-strom, atd.).

cTreeHeader

Je odvozenou třídou od *cDStructHeader*. Navíc uchovává informace specifické pro stromovou DS (např. výška stromu).

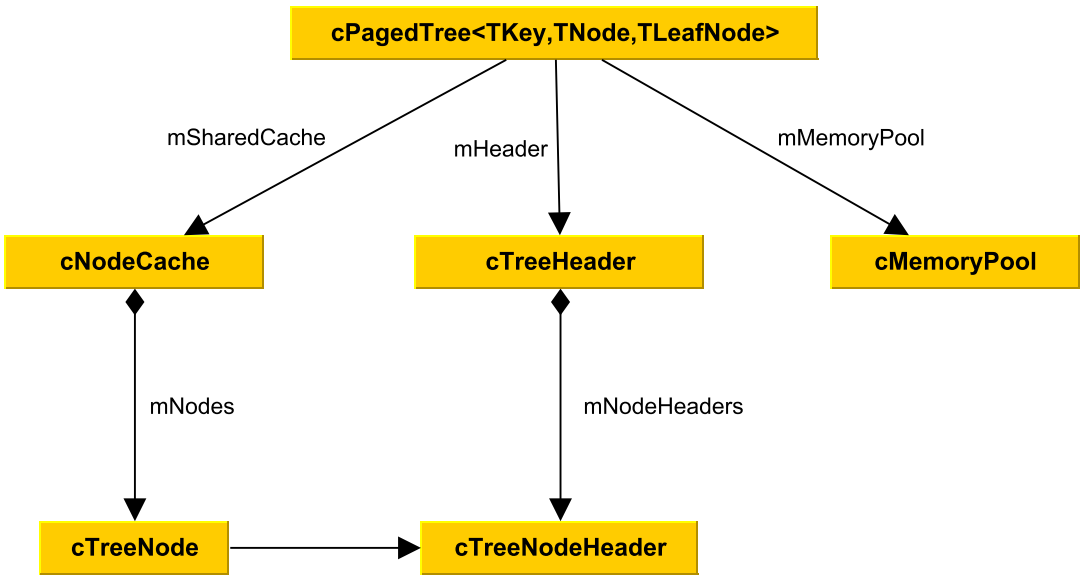
cTreeNodeHeader

Je odvozenou třídou od *cNodeHeader*. Navíc jsou zde uloženy offsety pro uspořádání dat v poli mData.

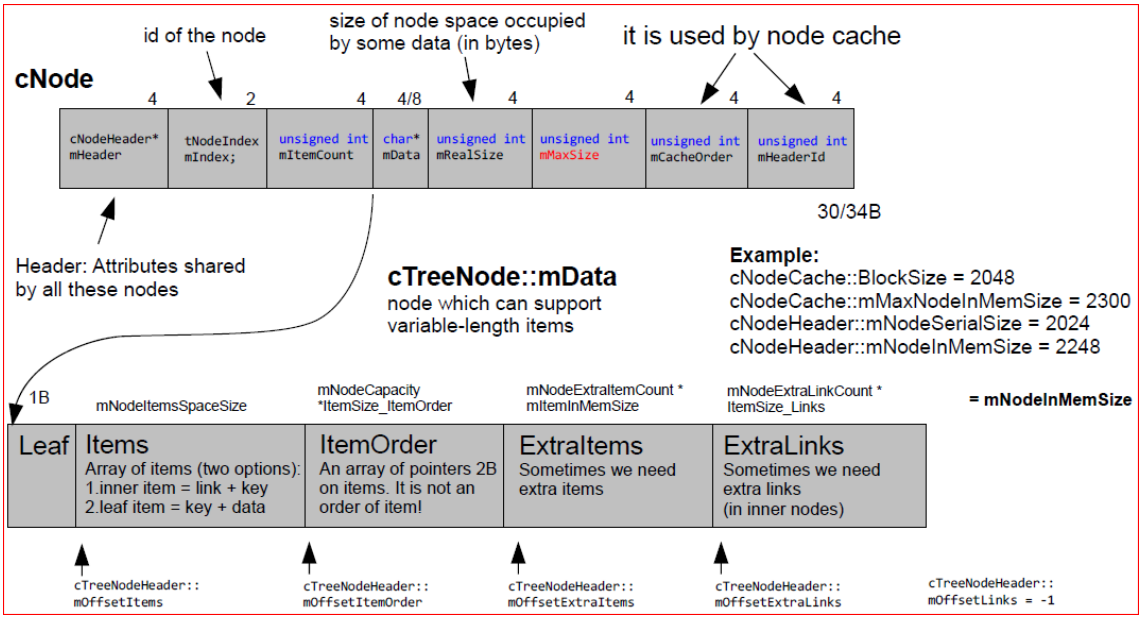
cTreeNode

Je odvozenonu třídou od *cNode*. Díky ofsetům ze třídy *cTreeNodeHeader* můžeme pole mData rozdělit do několika segmentů, které jsou patrné na obrázku 8, a které si nyní popíšeme.

1. *Leaf* - 1 bajtový příznak, který nám udává, zda se jedná o listový nebo vnitřní uzel.
2. *Items* - Segment prvků. Jedná se o nesetříděné pole, ve kterém jsou prvky fyzicky uloženy. V případě vnitřního uzlu jsou prvky reprezentovány jako odkaz



Obrázek 7: Schéma perzistentní stromové DS



Obrázek 8: Schéma uložení dat v uzlu.

(na podřízený uzel) + klíč. V případě listového uzlu jsou prvky reprezentovány jako klíč + data.

3. *ItemOrder* - Segment ukazatelů na prvky v segmentu Items. V tomto případě jde o setříděné pole.
4. *ExtraItems* - Segment prvků využívaný v případě potřeby.
5. *ExtraLinks* - Segment ukazatelů na sousední uzly. V případě vnitřního uzlu je uložen pouze ukazatel na nadřazený uzel. V případě listového uzlu jsou uloženy ukazatele na levý, pravý a nadřazený uzel.

4.4 Implementace sekvenčního pole

Implementace sekvenčního pole je přímo postavena nad základní strukturou tříd pro stránkované DS. Nyní se podíváme na odvozené třídy pro sekvenční pole.

cSequentialArray - Reprezentuje sekvenční pole jako takové.

cSequentialArrayHeader - Odvozená třída od *cDStructHeader*.

cSequentialArrayNodeHeader - Odvozená třída od *cNodeHeader*.

cSequentialArrayNode - Odvozená třída od *cNode*.

4.5 Implementace B+stromu

Implementace B+stromu je přímo postavena nad strukturou tříd pro stromové DS. Nyní se podíváme na odvozené třídy pro B+strom.

cCommonBpTree - Odvozená třída od *cPagedTree*.

cBpTreeHeader - Odvozená třída od *cDStructHeader*.

cBpTreeNodeHeader - Odvozená třída od *cTreeNodeHeader*.

cBpTreeNode - Odvozená třída od *cTreeNode*.

5 Rozšíření implementace B+stromu

Jak již bylo uvedeno v předchozí kapitole, tak implementační část této diplomové práce je rozšířením databázového rámce RadegastDB. Tento rámec nám ve výchozí implementaci B+stromu nabízí pouze operace pro vkládání a vyhledávání. Proto je potřeba jej rozšířit o operace mazání a aktualizace.

Ještě před tím než přejdeme k popisu samotné implementace, je potřeba zmínit některé vlastnosti, které nám rámec nad B+stromem nabízí a které se přímo týkají této části práce.

- **Způsob uložení dat**

  Rámec nám v tomto ohledu nabízí dvě možnosti.

  1. Uložení dat s pevnou délkou
  2. Uložení dat s proměnnou délkou

  Obě tyto možnosti uložení mají své výhody i nevýhody. V principu je práce s pevnou délkou dat jednodušší a rychlejší ale za to klade větší nároky na kapacitu úložiště. Oproti tomu je práce s proměnnou délkou dat složitější a pomalejší ale s pohledu kapacity je úspornější.

  Pokud se bavíme o jednodušší, resp. složitější práci máme tím na myslí jednodušší, resp. složitější implementaci, což se přímo týká i této části práce.

- **Duplicitní klíče**

  Další vlastností je možnost ukládat data s duplicitním klíčem. Je zřejmé, že i tato vlastnost se přímo týká implementace mazání a v další části textu narazíme na problémy s tím spojené.

Poznámka 5.1 Všechny tyto vlastnosti jsou volitelné a je možné je nastavit při vytváření databáze.

5.1 Implementace operace mazání

Z teoretických znalostí víme, že základem B+stromu jsou vnitřní a listové uzly. Tyto uzly jsou v našem rámci reprezentovány třídou *cTreeNode*, která je potomkem třídy *cNode*. Je tedy zřejmé, že tato třída tvoří jakési jádro pro práci s daty na té nejnižší úrovni. Proto i s popisem implementace operace mazání, začneme právě zde.

Než se pustíme do popisu jednotlivých algoritmů a funkcí, je potřeba si uvědomit jakým způsobem jsou data v uzlu uložena. K lepšímu pochopení způsobu uložení dat nám poslouží obrázek 8 z kapitoly 4.3. Z obrázku je patrné, že data uzlu jsou na nejnižší úrovni reprezentována jako pole znaků *(*char)*. Toto pole lze rozdělit do několika

segmentů pomocí offsetů, které jsou definovány v hlavičce uzlu, tedy ve třídě *cTreeNodeHeader*. Detailnější popis uvedených segmentů najdeme v kapitole 4.3. Pro zjednodušení budou tyto části[2] v následujícím textu považovány za samostatná pole.

Abychom mohli data fyzicky smazat potřebujeme zajistit, aby byla odstraněna z pole *Items* a *ItemOrder*.

5.1.1 Smazání prvku z pole *Items*

Pro smazání prvku z pole *Items* byla vytvořena funkce *RemoveItemPo* (viz algoritmus 1). Funkce zajistí smazání prvku, jak pro listový, tak pro vnitřní uzel.

Poznámka 5.2 Prvek je chápán jako dvojice hodnot *klíč* + *data* pro listový uzel a *odkaz* + *klíč* pro vnitřní uzel.

Před popisem samotného algoritmu je potřeba upozornit na skutečnost, že prvek je nahrazen posunutím celého bloku paměti, což je patrné z řádku 8 v algoritmu 1, kde je posunut celý blok paměti jehož velikost je vypočtená na řádku 7. Nedochází tedy k pouhému označení, že byl prvek smazán, ale k definitivnímu odstranění. Tento přístup má své výhody i nevýhody. Jednou s nepopíratelných výhod je jistě úspora paměti (úložiště). Ovšem zásadní nevýhodou je snížení výkonnosti při přesunech bloků paměti s čímž souvisí také potřebná aktualizace setříděného pole *ItemOrder*.

Popis algoritmu

- V první řadě je určena skutečná pozice v poli *mData*, která je vypočtena pomocí ofsetu pole *Items* a vstupního parametru funkce, který udává pozici prvku v poli *Items* (viz řádek 5 v algoritmu 1). Přičtení vypočtené pozice k ukazateli na pole *mData*, tak získáme skutečnou adresu prvku.
- Jak jsme již uvedli, tak prvek je odstraněn posunutím celého bloku paměti. V dalším kroku je tedy potřeba určit velikost této paměti (viz řádek 7 v algoritmu 1).
- V poslední řadě je proveden samotný přesun bloku paměti, čímž dojde k odstranění prvku z pole *Items* (viz řádek 8 v algoritmu 1).

5.1.2 Smazání prvku z pole *ItemOrder*

Pro smazání prvku z pole *ItemOrder* byla vytvořena funkce *RemoveItemOrder* (viz algoritmus 2).

Poznámka 5.3 V poli *ItemOrder* chápeme prvek jako odkaz na prvek z poli *Items*.

Smazání funguje obdobně jako v předchozím případě, tzn. prvek je smazán posunutím celého bloku paměti, což je patrné z řádku 5 v algoritmu 2.

[2]Kromě části *Leaf*, která představuje jedno bitový příznak určující zda se jedná o listový, resp. vnitřní uzel.

```
1 Function RemoveItemPo(PozicePrvkuVPoliItems)
      Result: Velikost mazaného prvku
2     mData ← Ukazatel na pole mData;
3     VelikostVšechPrvků ← Velikost všech prvků v uzlu;
4     OfsetPoleItems ← Offset na pole Items;
5     PoziceKlíče ← Pozice klíče vypočtená jako OfsetPoleItems +
      PozicePrvkuVPoliItems;
6     VelikostPrvku ← Velikost mazaného prvku;
7     VelikostPaměti ← Velikost paměti pro přesun vypočtená jako (OfsetPoleItems +
      VelikostVšechPrvků) - (PoziceKlíče + VelikostPrvku);
8     Blok paměti na adrese mData + PoziceKlíče + VelikostPrvku o velikosti
      VelikostPaměti přesuň na adresu mData + PoziceKlíče;
9     Vrátit VelikostPrvku;
```

Algorithm 1: Funkce cTreeNode::RemoveItemPo

Popis algoritmu

- V první řadě je určena skutečná pozice v poli *mData*, která je vypočtena pomocí ofsetu pole *ItemOrder* a vstupního parametru funkce, který udává pozici prvku v poli *ItemOrder* (viz řádek 3 v algoritmu 2). Přičtení vypočtené pozice k ukazateli na pole *mData*, tak získáme skutečnou adresu prvku v poli *ItemOrder*, tedy pořadí prvku.
- V dalším kroku určíme velikost paměti pro přesun (viz řádek 4 v algoritmu 2).
- V poslední řadě je proveden samotný přesun bloku paměti, čímž dojde k odstranění prvku z pole *ItemOrder* (viz řádek 5 v algoritmu 2).

```
1 Function RemoveItemOrder(PozicePrvkuVPoliItemOrder)
2     PočetPrvků ← Počet všech prvků v uzlu;
3     PrvekVPoliItemOrder ← Ukazatel na prvek v poli ItemOrder;
4     VelikostPaměti ← Velikost paměti pro přesun vypočtená jako PočetPrvků-
      (PozicePrvkuVPoliItemOrder + 1);
5     Blok paměti na adrese PrvekVPoliItemOrder +1 o velikosti VelikostPaměti přesun na
      adresu PrvekVPoliItemOrder;
```

Algorithm 2: Funkce cTreeNode::RemoveItemOrder

5.1.3 Smazání prvku z uzlu

Výše uvedené funkce nejsou z pohledu konzistence DS bezpečné. Jsou proto psány jako privátní funkce třídy *cTreeNode* a je potřeba k nim přistupovat s velkou opatrností.

Pro bezpečné smazání prvku v kontextu celého uzlu byla vytvořena veřejná funkce *Delete* (viz algoritmus 3). Tato funkce však využívá již zmíněných funkcí *RemoveItemPo* a *RemoveItemOrder*.

Návratové hodnoty funkce

- `DELETE_YES` - Signalizuje úspěšně smazání prvku.
- `DELETE_AT_THE_END` - Signalizuje, že byl smazán poslední prvek uzlu.
- `DELETE_NOTEXIST` - Signalizuje, že prvek v uzlu neexistuje.
- `DELETE_NOTEXIST_DATA` - Signalizuje, že prvek s daným klíčem existuje, ale neexistuje prvek v kombinaci klíč a data (případ povolení duplicitních klíčů).

Popis algoritmu

- V první řadě je potřeba zjistit pořadí zadaného klíče. V případě, že jsou povoleny duplicitní klíče, tak je potřeba najít pořadí dvojice klíč a data (viz řádek 2 až 6 v algoritmu 3).
- Pokud je prvek s daným klíčem, resp. klíčem a daty v uzlu nalezen zjistíme jeho pozici v poli *Items* (viz řádek 8 v algoritmu 3).
- Poté je prvek odstraněn z pole *Items* pomocí privátní funkce *RemoveItemPo* (viz řádek 9 v algoritmu 3).
- Následně je prvek odstraněn také z pole *ItemOrder* pomocí privátní funkce *RemoveItemOrder* (viz řádek 10 v algoritmu 3).
- Jelikož jsou prvky mazány posunutím celého bloku paměti, je potřeba aktualizovat odkazy v poli *ItemOrder* (viz řádek 11 až 15 v algoritmu 3).
- Dále provedeme aktualizaci informace o celkovém počtu prvků a volného místa (viz řádek 16 a 17 v algoritmu 3).
- Na konci je vrácena adekvátní návratová hodnota.

5.1.4 Přesun prvků mezi uzly

Vzhledem k tomu, že implementujeme operaci mazání, je potřeba počítat s přesuny mezi uzly. K tomuto účelu byla vytvořena funkce *MoveItems* (viz algoritmus 5). Tato funkce přijímá jako parametr ukazatel na zdrojový uzel, ze kterého se budou prvky přesouvat. Počet přesouvaných prvků je počítán, tak aby poměr prvků v obou uzlech byl pokud možno vyrovnaný. Snahou tedy je, aby výsledný strom byl co nejvyrovnanější.

Vzorec pro výpočet počtu prvků k přesunu je pro variantu s pevnou délkou klíče a dat následující:

```
1  Function Delete(Klíč, Data, PovolitDuplicitníKlíče)
       Result: Výsledek mazání (viz návratové hodnoty funkce)
2      if PovolitDuplicitníKlíče then
3          PořadíPrvku ← NajítPořadíPrvku(Klíč, Data);
4      else
5          PořadíPrvku ← NajítPořadíPrvku(Klíč);
6      end if
7      if Klíč a Data existují then
8          PozicePrvkuVPoliItems ← ZískejPoziciPrvkuVItems(PořadíPrvku);
9          VymažPrvekZItems(PozicePrvkuVPoliItems);
10         VymažPrvekZItemOrder (PořadíPrvku);
11         for i = 0 to CelkovýPočetPrvků −1 do
12             if ZískejPoziciPrvkuVItems(i) > PozicePrvkuVPoliItems then
13                 Aktualizovat prvek i v ItemOrder;
14             end if
15         end for
16         Aktualizovat CelkovýPočetPrvků;
17         Aktualizovat volné místo v uzlu;
18         if PořadíPrvku == CelkovýPočetPrvků −1 then
19             Vrať informaci, že byl smazán poslední prvek uzlu;
20         else
21             Vrať informaci, že prvek byl úspěšně smazán;
22         end if
23     else
24         Vrať informaci, že prvek neexistuje;
25     end if
```

Algorithm 3: Funkce cTreeNode::Delete

$$PocetPrvkuKPresunu = (CelkovyPocetPrvkuZdrojovehoUzlu - MinimalniPocetPrvkuUzlu + 1)/2$$

V případě variabilní délky klíče, resp. dat je výpočet o něco komplikovanější a je potřeba jej znázornit algoritmicky (viz algoritmus 4).

```
1  VelikostPrvkůKPřesunu ←
   (VelikostVšechPrvkůZdrojovéhoUzlu − MinimálníPočetPrvkůUzlu)/2;
2  if Zdrojový uzel je levým sousedem then
3  |  pořadíPrvku ← CelkovýPočetPrvkůZdrojovéhoUzlu − 1;
4  else
5  |  pořadíPrvku ← 0;
6  end if
7  while VelikostPrvkůKPřesunu > 0 do
8  |  VelikostPrvkůKPřesunu ←
   |  VelikostPrvkůKPřesunu − VelikostPrvkuZrojovéhoUzlu(pořadíPrvku);
9  |  if Zdrojový uzel je levým sousedem then
10 |  |  pořadíPrvku ← pořadíPrvku − 1;
11 |  else
12 |  |  pořadíPrvku ← pořadíPrvku + 1;
13 |  end if
14 |  PočetPrvkůKPřesunu ← PočetPrvkůKPřesunu + 1;
15 end while
```

Algorithm 4: Výpočet počtu prvků k přesunu pro variabilní délku klíče

Popis algoritmu

- V první řadě je proveden výpočet počtu prvků k přesunu, který jsme si již zmínili v předchozím textu (viz 2 řádek v algoritmu 5).
- Poté je stanoveno pořadí na jaké budou prvky vkládány a pořadí ve zdrojovém uzlu, ze kterého budou prvky přesouvány (viz řádek 2 až 10 v algoritmu 5). Pokud je zdrojový uzel levým sousedem, pak je potřeba mimo jiné vytvořit potřebný prostor v poli *ItemOrder* (viz řádek 6 v algoritmu 5).
- Dále jsou jednotlivé prvky ve zdrojovém uzlu přesunuty do aktuálního uzlu. Prvek je následně ze zdrojového uzlu smazán (viz řádek 11 až 14 v algoritmu 5).
- Na závěr je aktualizován počet prvků a volné místo v aktuálním i zdrojovém uzlu (viz řádek 15 a 16 v algoritmu 5).

```
1  Function MoveItems(ZdrojovýUzel)
2      PočetPrvkůKPřesunu ← Počet prvků k přesunu;
3      if Zdrojový uzel je levým sousedem then
4          PořadíVkládanéhoPrvku ← 0;
5          PořadíMazanéhoPrvku
           ← CelkovýPočetPrvkůZdrojovéhoUzlu − PočetPrvkůKPřesunu;
6          Vytvořit v poli ItemOrder místo pro PočetPrvkůKPřesunu prvků;
7      else
8          PořadíVkládanéhoPrvku ← CelkovýPočetPrvků;
9          PořadíMazanéhoPrvku ← 0;
10     end if
11     for i ← 1 to PočetPrvkůKPřesunu do
12         Přesunout prvek PořadíMazanéhoPrvku ze zdrojového uzlu na pozici
           PořadíVkládanéhoPrvku aktuálního uzlu;
13         Smazat prvek PořadíMazanéhoPrvku ze zdrojového uzlu;
14     end for
15     Aktualizovat počet prvků a volné místo ve zdrojovém uzlu;
16     Aktualizovat počet prvků a volné místo v aktuálním uzlu;
```

Algorithm 5: Funkce cTreeNode::MoveItems

5.1.5 Sloučení uzlů

Další operací, kterou je potřeba při mazání prvků zajistit, je sjednocení uzlů. K tomu byla ve třídě *cTreeNode* vytvořena funkce *Merge* (viz algoritmus 6). Parametrem funkce je opět ukazatel na zdrojový uzel, se kterým se bude aktuální uzel slučovat.

Před samotným popisem algoritmu je potřeba ještě upozornit na skutečnost, že zdrojový uzel je vždy pravým sousedem aktuálního uzlu. V principu to znamená, že zdrojový uzel zaniká.

Popis algoritmu

- V první řadě jsou přeneseny všechny prvky ze zdrojového uzlu na pořadí posledního prvku + 1 aktuálního uzlu (viz řádek 2 až 4 v algoritmu 6). Pořadí je dáno tím, že zdrojový uzel je vždy pravým sousedem aktuálního uzlu.
- V další řadě jsou aktualizovány informace o počtu prvků a volném místě v uzlu.

Poznámka 5.4 Jak je možné vidět, tak algoritmus 6 je stejný, jak pro implementaci listového uzlu, tak pro implementaci vnitřního uzlu. Rozdíl je pouze v reprezentaci prvku.

```
1 Function Merge(ZdrojovýUzel)
2     for i ← 1 to CelkovýPočetPrvkůZdrojovéhoUzlu do
3         Přesunout prvek i ze zdrojového uzlu na pozici
          CelkovýPočetPrvkůAktuálníhoUzlu + i aktuálního uzlu;
4     end for
5     Aktualizovat počet prvků a volné místo v aktuálním uzlu;
```

Algorithm 6: Funkce cTreeNode::Merge

5.1.6 Smazání prvku ze stromu

Všechny funkce, které jsme si doposud ukázali nám zajišťují operace mazání, přesun a sloučení nad jedním konkrétním uzlem. Naším cílem však je zajistit smazání prvku z celého B+stromu. Je tedy patrné, že všechny operace, které byli implementovány v rámci uzlu budou implementovány také v rámci celého B+stromu.

Z kapitoly 4 víme, že stromová DS je reprezentována třídou *cPagedTree*. Jednou z jejich odvozených tříd je třída *cCommonBpTree*, která reprezentuje B+strom. Tato třída nám tedy posloužila k implementaci výše uvedených operaci. Konkrétně pro smazání prvku byla ve třídě *cCommonBpTree* vytvořena funkce *Delete*.

Scénáře při mazání prvku Jednotlivé scénáře, které mohou při mazání prvku z B+stromu nastat si demonstrujeme na příkladu 5.1.

Příklad 5.1
Předpokládejme, že máme B+strom z obrázku 9. Minimální počet prvků v uzlu je 2 a maximální počet prvků v uzlu je 5. Prvky jsou ve vnitřním uzlu uspořádány jako *odkaz:klíč* oddělené znakem "|". V listovém uzlu jsou prvky uspořádány jako *klíč:data* oddělené znakem "|". ■

Nyní ke každému scénáři uvedeme konkrétní příklad pro náš vzorový B+strom z příkladu 5.1 při kterém tento scénář nastane.

1. **Prvek je úspěšně smazán a není potřeba dalších změn v DS**

   Příkladem může být smazání prvku s hodnotou klíče 20. Prvek je nalezen v Uzlu2 a poté je smazán. Dále již není potřeba jakýchkoli změn.

2. **Prvek s daným klíčem neexistuje**

   Příkladem může být pokus o smazání prvku s klíčem 4.

3. **Prvek s daným klíčem a daty neexistuje**

   Jedná se o případ, kdy jsou povoleny duplicitní hodnoty klíče. Příkladem může být pokus o smazání prvku s klíčem 5 a s hodnotou dat "G". Ikdyž se prvek s klíčem 5 nachází v DS dvakrát (5:F a 5:H), tak kombinace hodnot klíče 5 a dat "G"neexistuje.

4. **Prvek je smazán, ale je potřeba aktualizace klíčů nadřazených vnitřních uzlů**

   Příkladem může být smazání prvku s hodnotou klíče 8. Jak je patrné, tak klíč s hodnotou 8 je posledním prvkem uzlu Uzel1. V takovém případě je potřeba provést aktualizaci prvku v nadřazeném vnitřním uzlu, který se na změněný uzel odkazuje. V našem příkladu vidíme, že nadřazený vnitřní uzel Uzlu1 je Uzel4. Dále můžeme vidět, že na Uzel1 se odkazuje první prvek. První prvek je tvořen dvojicí odkaz (s hodnotou Uzel1) a klíč (s hodnotou 8). Právě hodnotu klíče 8 je potřeba změnit na novou hodnotu 2.

   Je potřeba upozornit na to, že stejný algoritmus platí i při změně, resp. smazání prvku ve vnitřním uzlu. Propagace změn prvků nadřazeného vnitřního uzlu pak postupuje až ke kořeni.

5. **Prvek je smazán, ale je potřeba přesun prvků mezi uzly**

   Příkladem může být smazání prvku s hodnotou klíče 12. Po smazání obsahuje Uzel7 pouze 1 prvek, což je pod povoleným minimem. Řešením je buď sloučení se sousedním uzlem nebo přesun prvků ze sousedního uzlu. Vzhledem k tomu, že sousední uzel Uzel2 obsahuje maximální počet prvku, tak sloučení není možné. Přesun však možný je. Z uzlu Uzel2 budou tedy přesunuty prvky s klíčem 15 a 16.

   Opět je potřeba upozornit na to, že stejný algoritmus platí i pro vnitřní uzly.

6. **Prvek je smazán, ale je potřeba sloučení dvou uzlů**

   Příkladem může být smazání prvku s hodnotou klíče 2. Stejně jako v předchozím příkladě, tak i zde je po smazání nedostatečné množství prvků, tzn. je potřeba provést sloučení se sousedním uzlem nebo přesun ze sousedního uzlu. Sousední uzel Uzel5 obsahuje 3 prvky, takže je možné provést sloučení, které má v naší implementaci přednost před přesunem. Po sloučení zůstane pouze Uzel1 a Uzel5 zanikne.

   Stejný algoritmus opět platí také pro vnitřní uzly.

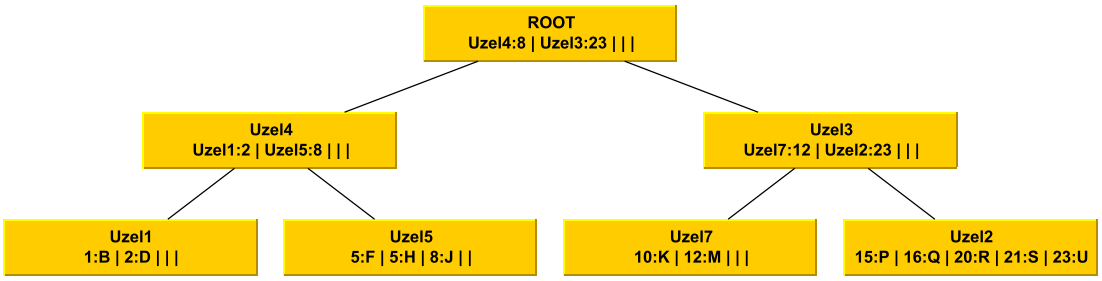
7. **Prvek je smazán, ale je potřeba snížení výšky stromu**

   Příkladem může být opět smazání prvku s hodnotou klíče 2. Po smazání prvku dojde ke sloučení Uzlu1 a Uzlu5. Poté je smazán druhý prvek z Uzlu4. Po smazání zůstane v uzlu nedostatečný počet prvku a proto dojde ke sloučení s uzlem Uzel3. Po sloučení zůstane pouze Uzel4. Vzhledem k tomu, že v tuto chvíli má kořenový uzel pouze jednoho potomka stane se novým kořenem Uzel4.

Popis algoritmu Jak je z algoritmu 7 patrné, tak je možné algoritmus mazání rozdělit do 3 částí:

1. **Dopředný průchod stromem - vyhledání a smazání prvku** (viz algoritmus 8)

   V této části je prvek vyhledán v konkrétním listovém uzlu a následně je smazán. Poté je vyhodnocen výsledek smazání.



Obrázek 9: Vzorový B+strom k demonstraci scénářů

V případě, že byl smazán poslední prvek uzlu, přejde algoritmus do 3 části, tedy zpětného průchodu stromem k úpravě klíče nadřazeného uzlu.

Pokud je po smazání v uzlu nedostatečné množství prvků, dojde buď ke sloučení uzlů nebo k přesunu prvků mezi uzly. V případě, že jsou sloučeny dva sousední uzly přejde algoritmus do části pro zpětný průchod stromem ke slučování. Pokud však dojde k přesunu prvků, tak algoritmus přejde do části pro zpětný průchod stromem k úpravě klíče nadřazeného uzlu. Může se také stát, že se nenajde vhodný sousední uzel ani pro sloučení ani pro přesun. V takovém případě je algoritmus ukončen s kladným výsledkem.

Se záporným výsledkem algoritmus skončí v případě, kdy se prvek v listovém uzlu nenajde.

Poslední možností je, že prvek je smazán a není potřeba dalších úprav DS. V takovém případě algoritmus končí s kladným výsledkem.

2. **Zpětný průchod stromem - slučování** (viz algoritmus 9)

   V první řadě je načten nadřazený uzel (předek) podřízeného uzlu (v případě prvního průchodu jde o listový uzel). Poté je aktualizován klíč nadřazeného uzlu u prvku, který ukazuje na podřízený uzel. Dále je možné smazat prvek vnitřního uzlu, který ukazoval na uzel, který při sloučení podřízených uzlů zanikl. Po smazání se opět vyhodnotí výsledek.

   Pokud je po smazání v uzlu nedostatečné množství prvků, dojde opět buď ke sloučení nebo k přesunu. Pokud dojde ke sloučení, algoritmus pokračuje dále ve zpětném průchodu stromem ke slučování. V případě, že dojde k přesunu prvků, tak algoritmus stejně jako u listového uzlu přejde do části pro zpětný průchod stromem k úpravě klíče nadřazeného uzlu.

   Pokud algoritmus dojde ke kořeni, který má po smazání pouze jeden prvek dojde ke snížení výšky stromu.

   Algoritmus pokračuje v průchodu stromem dokud je potřeba sloučení uzlů nebo dokud nedojde ke kořeni. V takovém případě algoritmus končí a s kladným výsledkem.

3. **Zpětný průchod stromem - úprava klíče nadřazeného uzlu** (viz algoritmus 10)

   V první řadě je načten nadřazený uzel (předek) podřízeného uzlu (v případě prvního průchodu jde o listový uzel). Poté je aktualizován klíč nadřazeného uzlu u prvku, který ukazuje na podřízený uzel.

   Pokud je aktualizován klíč posledního prvku v uzlu, pokračuje algoritmus dále v průchodu stromem. V opačném případě algoritmus končí s kladným výsledkem.

```
1 Function Delete(Klíč, Data)
2 |  Dopředný průchod stromem - vyhledání a smazání prvku Klíč + Data;
3 |  Zpětný průchod stromem - slučování a snížení výšky stromu;
4 |  Zpětný průchod stromem - úprava klíče nadřazených vnitřních uzlů;
```

Algorithm 7: Funkce cCommonBpTree::Delete

```
1  AktuálníUzel ← NačístUzel(ČísloKořenovéhoUzlu);
2  while AktuálníUzel je vnitřní uzel do
3  |  ČísloUzlu ← NajítPrvekAktuálníhoUzlu(Klíč);
4  |  AktuálníUzel ← NačístUzel(ČísloUzlu);
5  end while
6  SmazatPrvekAktuálníhoUzlu(Klíč, Data);
7  if Smazání proběhlo úspěšně then
8  |  Vratit informaci o úspěšném smazání;
9  else if Mazaný prvek neexistuje then
10 |  Vratit informaci o neúspěšném smazání;
11 else if AktuálníUzel nemá dostatek prvků then
12 |  SousedníUzel ← NačístSousedníUzel(AktuálníUzel);
13 |  if AktuálníUzel a SousedníUzel je možné sloučit then
14 |  |  Sloučit(AktuálníUzel, SousedníUzel);
15 |  else if SousedníUzel je vhodný k přesunu prvků then
16 |  |  PřesunoutPrvky(AktuálníUzel, SousedníUzel);
17 |  else
18 |  |  Vratit informaci o úspěšném smazání;
19 |  end if
20 end if
```

Algorithm 8: Dopředný průchod stromem - vyhledání a smazání prvku

5.1.7 Přesun prvků mezi uzly v rámci stromu

Pro potřeby přesunu prvků byly ve třídě *cCommonBpTree* vytvořeny dvě třídy.

```
1  while Proběhlo sloučení do
2  |  PodřízenýUzel ← AktuálníUzel;
3  |  AktuálníUzel ← NačístNadřazenýUzel(AktuálníUzel);
4  |  Pořadí ← NajítPořadíPodřízenéhoUzlu(AktuálníUzel, PodřízenýUzel);
5  |  AktualizovatKlíčAktuálníhoUzlu(Pořadí, PodřízenýUzel);
6  |  SmazatPrvekAktuálníhoUzlu(Pořadí + 1);
7  |  if AktuálníUzel je kořen a počet prvků = 1 then
8  |  |  Snížení výšky stromu;
9  |  |  Vrať informaci o úspěšném smazání;
10 |  else if AktuálníUzel nemá dostatek prvků then
11 |  |  SousedníUzel ← NačístSousedníUzel(AktuálníUzel);
12 |  |  if AktuálníUzel a SousedníUzel je možné sloučit then
13 |  |  |  Sloučit(AktuálníUzel, SousedníUzel);
14 |  |  else if SousedníUzel je vhodný k přesunu prvků then
15 |  |  |  PřesunoutPrvky(AktuálníUzel, SousedníUzel);
16 |  |  end if
17 |  else
18 |  |  Vrať informaci o úspěšném smazání;
19 |  end if
20 end while
```

Algorithm 9: Zpětný průchod stromem - slučování a snížení výšky stromu

```
1  while Byl změněn poslední prvek uzlu do
2  |  PodřízenýUzel ← AktuálníUzel;
3  |  AktuálníUzel ← NačístNadřazenýUzel(AktuálníUzel);
4  |  Pořadí ← NajítPořadíPodřízenéhoUzlu(AktuálníUzel, PodřízenýUzel);
5  |  AktualizovatKlíčAktuálníhoUzlu(Pořadí, PodřízenýUzel);
6  end while
7  Vrať informaci o úspěšném smazání;
```

Algorithm 10: Zpětný průchod stromem - úprava klíče nadřazených vnitřních uzlů

1. **MoveItemsBetweenLeafNodes** - pro přesun prvků mezi listovými uzly
2. **MoveItemsBetweenInnerNodes** - pro přesun prvků mezi vnitřními uzly

Obě funkce se liší pouze v datových typech vstupních parametrů, proto další popis algoritmu bude obecný pro obě funkce.

Popis algoritmu

- V první řadě je pro aktuální uzel volána funkce *MoveItems* třídy *cTreeNode*, kterou jsme si popsali již dříve.
- Vzhledem k tomu, že při přesunu dojde ke změně posledního prvku jednoho z uzlů je potřeba klíč tohoto prvku uložit pro následující aktualizaci klíče v nadřazeném vnitřním uzlu (viz algoritmus 10). Proto zjistíme, zda je zdrojový uzel levým nebo pravým sousedem aktuálního uzlu.
- V případě, že zdrojový uzel je levým sousedním uzlem, uložíme klíč jeho posledního prvku.
- V opačném případě, tzn. pokud je zdrojový uzel pravým sousedním uzlem, uložíme klíč posledního prvku aktuálního uzlu.

5.1.8 Sloučení uzlů v rámci stromu

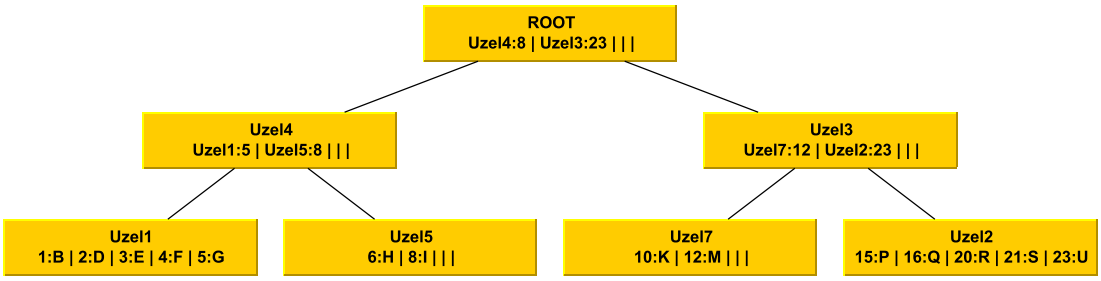
Dále ve třídě *cCommonBpTree* vznikly dvě funkce, které nám zajišťují sloučení uzlů.

1. **MergeLeafNodes** - pro sloučení listových uzlů
2. **MergeInnerNodes** - pro sloučení vnitřních uzlů

V tomto případě už je implementace obou funkcí odlišná a proto další popis algoritmu bude pro obě funkce zvlášť.

Popis algoritmu funkce MergeLeafNodes

- V první řadě zjistíme, zda je sousední uzel levým nebo pravým sousedem.
- V případě levého sousedního uzlu, uložíme sousední uzel jako levý a aktuální uzel jako pravý.
- V opačném případě uložíme aktuální uzel jako levý a sousední jako pravý.
- Provedeme sloučení z pravého uzlu do levého, tzn. pravý uzel zanikne.
- Odkaz pravého uzlu na svého pravého souseda uložíme jako odkaz levého uzlu na svého pravého souseda.
- Uložíme klíč posledního prvku levého uzlu.
- Pokud se levý uzel odkazuje na svého nového pravého souseda (viz bod 5), je potřeba upravit odkaz tohoto souseda pro levého souseda.



Obrázek 10: Vzorový B+strom k demonstraci problémů

Popis algoritmu funkce MergeInnerNodes

- V první řadě zjistíme, zda je sousední uzel levým nebo pravým sousedem.
- V případě levého sousedního uzlu, uložíme sousední uzel jako levý a aktuální uzel jako pravý.
- V opačném případě uložíme aktuální uzel jako levý a sousední jako pravý.
- Provedeme sloučení z pravého uzlu do levého, tzn. pravý uzel zanikne.
- Uložíme klíč posledního prvku levého uzlu.

5.1.9 Problémy při implementaci operace mazání

Během implementace bylo potřeba řešit netriviální množství problémů. Příčiny těchto problémů byli dvojího typu.

1. Problémy týkající se databázového rámce a jeho vlastností a omezení.
2. Problémy týkající se DS B+stromu a jeho vlastností a omezení.

Nyní se zaměříme na některé zásadní problémy a ukážeme si jejich řešení při implementaci.

1. **Problém při slučování a přenosu prvků**

   Jeden z prvních problémů, které bylo potřeba řešit, nastal při implementaci přenosu prvků mezi sousedními uzly a následně při implementaci sloučení dvou sousedních uzlů. Tento problém můžeme zařadit do kategorie problémů týkajících se DS.

   Podstatou problému je sloučení, resp. přenos prvků dvou sousedních uzlů, které mají odlišný nadřazený uzel (předka). Problém si demonstrujeme na příkladu 5.2.

Příklad 5.2
Předpokládejme, že máme B+strom z obrázku 10. Nyní se pokusíme smazat prvek s hodnotou klíče 12. Je zřejmé, že po smazání nebude mít uzel Uzel7 dostatečné množství prvků. Proto bude potřeba provést sloučení se sousedním uzlem nebo přenos prvků ze sousedního uzlu. V našem příkladu má uzel Uzel7 dva sousedy, tedy uzel Uzel5 a uzel Uzel2. Vidíme, že uzel Uzel2 má maximální počet prvků, takže není možné provést sloučení. Ovšem uzel Uzel5 má pouze 2 prvky, což nám umožňuje provést sloučení. Po sloučení uzlu Uzel5 a uzlu Uzel7 nám zůstane uzel Uzel5. Uzel Uzel7 zanikne. Protože byl změněn poslední prvek uzlu Uzel5 bude potřeba provést aktualizaci prvku Uzel5:8 v uzlu Uzel4 a následně aktualizaci prvku Uzel4:8 v uzlu ROOT. Zároveň byl také odstraněn uzel Uzel7 a proto je potřeba odstranit prvek Uzel7:12 v uzlu Uzel3. Důsledkem toho pak bude sloučení uzlů Uzel4 a Uzel3. Zůstane jeden uzel Uzel4, který se pak stane kořenem. ■

Z příkladu je zřejmé, že implementace takového případu sloučení, resp. přenosu prvků by byla komplikovaná (dva zpětné průchody stromem).

Řešením byla podmínka, která nám zajistí sloučení, resp. přenos prvků dvou sousedních uzlů se stejným nadřazeným uzlem.

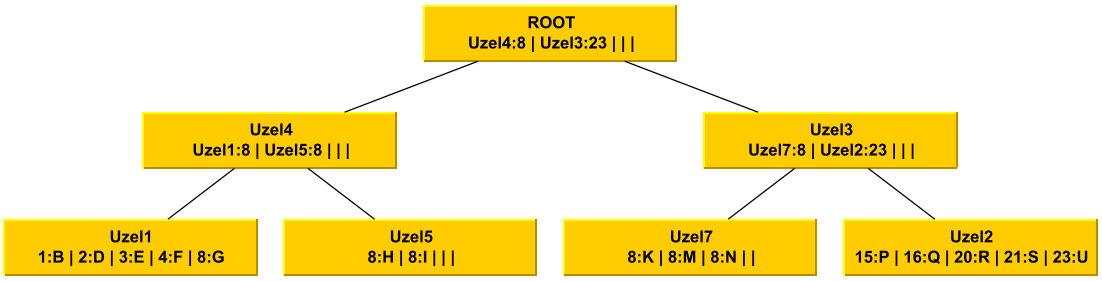
2. **Problém při aktualizaci klíče nadřazeného vnitřního uzlu**

   Další problém také můžeme zařadit do kategorie týkající se DS a opět souvisí se slučováním a přenosem prvků. Problém si demonstrujeme na příkladu 5.3.

   Příklad 5.3
   Předpokládejme, že máme B+strom z obrázku 10. Nyní se pokusíme smazat prvek s hodnotou klíče 8. Je zřejmé, že po smazání nebude mít uzel Uzel5 dostatečné množství prvků. Proto bude potřeba provést sloučení se sousedním uzlem nebo přenos prvků ze sousedního uzlu. V našem příkladu má uzel Uzel5 dva sousedy, tedy uzel Uzel1 a uzel Uzel7. Z řešení předchozího problému víme, že uzel Uzel7 není vhodný ke sloučení, protože má odlišný nadřazený uzel (předka). Uzel Uzel1 má zase maximální počet prvků, což opět neumožňuje sloučení. Vidíme ale, že je možné provést přenos prvků z uzlu Uzel1. Po přenosu prvků bude mít uzel Uzel1 prvky 1:B, 2:D a 3:E a uzel Uzel5 bude mít prvky 4:F, 8:I a 6:H. Je patrné, že došlo ke změně posledního prvku uzlu Uzel1 a proto je potřeba provést aktualizaci prvku Uzel1:5 v uzlu Uzel4. ■

   Na příkladu jsme mohli vidět situaci, kdy pro přenos prvků byl použit levý sousední uzel aktuálního uzlu (uzlu ze kterého je mazaný prvek). V takovém případě není pro zpětný průchod stromem použita stejná cesta, jako při dopředném průchodu.

   Pokud použiji výše uvedený příklad, tak pro dopředný průchod stromem byla pro nalezení prvku použita cesta ROOT $\rightarrow$ Uzel4 $\rightarrow$ Uzel5. Ovšem pro zpětný průchod stromem je potřeba použít cestu Uzel1 $\rightarrow$ Uzel4 $\rightarrow$ ROOT. Důvodem je změna posledního prvku uzlu Uzel1. Stejná situace také nastává při sloučení aktuálního uzlu



Obrázek 11: Vzorový B+strom k demonstraci problému s duplicitními klíči

s levým sousedním uzlem. V takovém případě aktuální uzel zaniká a zůstává levý sousední uzel. Proto je opět potřeba změnit cestu pro zpětný průchod stromem.

V implementaci se při dopředném průchodu stromem cesta zaznamenává do pole, ve kterém je uloženo pořadí prvků, kterými bylo potřeba projít. Cesta dopředného průchodu z příkladu by byla (0,1).

Řešení tohoto problému je následující. Pokud dojde ke sloučení s levým sousedním uzlem nebo k přenosu prvků z levého sousedního uzlu, je poslední prvek našeho pole dekrementován o jedna. Cesta zpětného průchodu z příkladu by pak vypadla následovně (0,0).

3. **Problém s duplicitními klíči**

   Další problém se týká samotné implementace rámce a v jeho podpory duplicitních klíčů. Problém si opět demonstrujeme na příkladu 5.4.

   Příklad 5.4
   Předpokládejme, že máme B+strom s povolenými duplicitními klíči z obrázku 11. Nyní se pokusíme smazat prvek s hodnotou klíče 8 a daty K. ■

   Z příkladu 5.4 je patrné, že dopředný průchod stromem nebude směřován k uzlu Uzel7, jak bychom potřebovali ale k uzlu Uzel1, který rovněž obsahuje prvek s klíčem 8. Z toho plynou následující dva problémy:

   (a) Vyhledání správného listového uzlu

   (b) Změna cesty dopředného průchodu stromem

   Pro vyhledání správného (z příkladu Uzel7) bylo do implementace potřeba přidat podporu sekvenčního průchodu listových uzlů, resp. sousedních uzlů aktuálního uzlu. Pokud se vrátíme k našemu příkladu 5.4, tak díky sekvenčnímu průchodu dojdeme až k uzlu Uzel7, který obsahuje mazaný prvek 8:K.

   Po nalezení správného uzlu nastává ale další problém, kterého si rovněž můžeme všimnout na našem příkladu 5.4. Řekli jsme si, že algoritmus dopředného průchodu

nalezne uzel Uzel1, který obsahuje klíč s hodnotou 8. Cesta dopředného průchodu je tedy (0,0). Ovšem po nalezení správného uzlu Uzel7 je potřeba cestu dopředného průchodu změnit na (1,0). Pro vyřešení tohoto problému bylo do implementace přidána aktualizace cesty dopředného průchodu.

4. **Problém při smazání posledního prvku uzlu**

   Další problém se týká DS. K demonstraci tohoto problému využijeme stejný příklad 5.3 z problému 2. Z příkladu 5.3 víme, že po odstranění prvku s klíčem 8 dojde ke sloučení uzlu "Uzel1"a uzlu "Uzel5", čímž je změněn poslední prvek uzlu "Uzel1". Je ale zřejmé, že pokud odstraníme prvek s klíčem 8 pak je změněn také poslední prvek uzlu "Uzel5". Z toho vyplývá, že pro zpětný průchod stromem musí být použity dvě cesty:

   (a) Uzel1 $\rightarrow$ Uzel4 $\rightarrow$ ROOT
   (b) Uzel5 $\rightarrow$ Uzel4 $\rightarrow$ ROOT

   V implementace bylo potřeba pro tento případ zajistit dva zpětné průchody.

5. **Problém s proměnnou délkou klíče a dat**

   Další problém se rovněž týká implementace rámce a jeho podpory uložení dat s proměnnou délkou. Pokud je proměnná délka dat povolena, mazání v některých případech způsobuje pád celé testovací aplikace. Tento problém se až doposud nepodařilo odstranit.

5.2 Implementace operace aktualizace

Operace aktualizace je oproti mazání v principu jednodušší. Důsledkem je jednodušší implementace a v konečném důsledku rychlejší zpracování operace. Důvodem jsou následující fakta:

1. Není potřeba zpětného průchodu stromem.
2. Není potřeba aktualizovat pořadí prvku v poli *ItemOrder*.
3. Týká se pouze listových uzlů.

S popisem opět začneme ve třídě *cTreeNode*, kde byli vytvořeny funkce pro aktualizaci prvku v rámci uzlu.

5.2.1 Aktualizace prvku v poli *Items*

Pro aktualizaci prvku v poli *Items* byla vytvořena funkce *ChangeDataPo* (viz algoritmus 11).

Popis algoritmu

- V první řadě získáme ukazatel na klíč, resp. prvek. K tomu nám poslouží pozice prvku v poli Items, kterou přičteme k ukazateli na pole *mData* a offsetu pro pole *Items*.
- Zjistíme velikost klíče.
- Zjistíme velikost dat. Zde se výpočet liší v závislosti na tom zda je povolena proměnná nebo pevná délka dat.
- Data jsou nahrazena zkopírováním bloku paměti.

```
1 Function ChangeDataPo(PozicePrvkuVPoliItems, Data)
      Result: Velikost aktualizovaného prvku
2     Klíč ← ZískatUkazatelNaKlíč(PozicePrvkuVPoliItems);
3     VelikostKlíče ← ZískatVelikostKlíče(Klíč);
4     VelikostDat ← ZískatVelikostDat(Data);
5     Blok paměti na adrese Data o velikosti VelikostDat zkopíruj na adresu Klíč +
      VelikostKlíče;
```

Algorithm 11: Funkce cTreeNode::ChangeDataPo

5.2.2 Aktualizace prvku v rámci uzlu

Výše uvedená funkce není z pohledu konzistence DS bezpečná. Proto je psána jako privátní funkce třídy *cTreeNode* a je potřeba k ní přistupovat s velkou opatrností.

Pro bezpečnou aktualizaci prvku v kontextu celého uzlu byla ve třídě *cTreeNode* vytvořena veřejná funkce *Update* (viz algoritmus 12).

Návratové hodnoty funkce

- `UPDATE_YES` - Signalizuje úspěšnou aktualizaci prvku.
- `UPDATE_NOTEXIST` - Signalizuje, že prvek s daným klíčem neexistuje.
- `UPDATE_NOTEXIST_DATA` - Signalizuje, že prvek s daným klíčem existuje ale neexistuje prvek v kombinaci klíč a data (případ s povolenými duplicitními klíči).

Popis algoritmu

- V první řadě zjistíme logické pořadí prvku. Zde je potřeba rozlišit, zda jsou povoleny duplicitní klíče nebo ne. Pokud duplicitní klíče povoleny nejsou najdeme pořadí prvku s daným klíčem. K tomu nám poslouží funkce *FindOrder*. Pokud duplicitní klíče povoleny jsou, je potřeba najít pořadí prvku s daným klíčem a daty.

Pro tento účel bylo potřeba implementovat přetíženou funkci *FindOrder*, která je oproti standardní funkci *FindOrder* rozšířena o parametr pro data, což je také patrné v první podmínce algoritmu 12.

- Poté ověříme zda prvek existuje. K tomu nám poslouží návratová hodnota výše uvedené funkce *FindOrder*, která v případě neexistence prvku vrací určitou konstantní hodnotu.
- Pokud prvek existuje zjistíme jeho fyzickou pozici v poli *Items*. Na této pozici je potřeba provést aktualizaci dat. K tomu nám poslouží výše popsaná funkce *ChangeDataPo*. Poté je možné vrátit informaci o úspěšné aktualizaci (viz návratová hodnota `UPDATE_YES`).
- V případě neexistence prvku je vrácena informace o tom, že prvek neexistuje. Pokud nejsou povoleny duplicitní klíče jde vždy o informaci, že nebyl nalezen klíč (viz návratová hodnota `UPDATE_NOTEXIST`). V opačném případě může jít o informaci, že byl nalezen klíč ale nebyla nalezena data (viz návratová hodnota `UPDATE_NOTEXIST_DATA`).

```
1  Function Update(Klíč, PůvodníData, NováData, PovolitDuplicitníKlíče)
       Result: Výsledek aktualizace
2      if PovolitDuplicitníKlíče then
3          PořadíPrvku ← NajítPořadíPrvku(Klíč, PůvodníData);
4      else
5          PořadíPrvku ← NajítPořadíPrvku(Klíč);
6      end if
7      if Klíč existuje nebo Klíč a PůvodníData existuje then
8          PozicePrvkuVPoliItems ←
           ZískatPoziciPrvkuVPoliItems(PořadíPrvku);
9          NahraditDataNaPozici(PozicePrvkuVPoliItems, NováData);
10         Vrátit informaci, že data prvku byla úspěšně nahrazen;
11     else
12         if Klíč neexistuje then
13             Vrátit informaci, že prvek s klíčem neexistuje;
14         else
15             Vrátit informaci, že prvek s klíčem a daty neexistuje;
16         end if
17     end if
```

Algorithm 12: Funkce cTreeNode::Update

5.2.3 Aktualizace prvku v rámci stromu

Doposud jsme si ukázali funkce, které nám zajišťují aktualizaci dat v rámci listového uzlu. Je však patrné, že je potřeba implementovat funkce i na vyšší úrovni, tedy nad celou DS.

Nyní se tedy přesuneme o úroveň výše a popíšeme si implementaci aktualizace v rámci celého stromu. Jak už víme z kapitoly 5.1, tak B+strom je v našem rámci reprezentován třídou *cCommonBpTree*. Pro potřeby aktualizace byla v této třídě vytvořena funkce *Update* (viz algoritmus 13).

Popis algoritmu

- Jak je s algoritmu 13 patrné, tak se v první řadě provádí dopředný průchod stromem za účelem nalezení listového uzlu obsahujícího zadaný klíč.
- Po nalezení listového uzlu je provedena samotná aktualizace, kdy je volána funkce *Update* třídy *cTreeNode*.
- Poté je vyhodnocena návratová hodnota funkce *Update* třídy *cTreeNode*. V případě návratové hodnoty `UPDATE_YES` je vrácena informace o úspěšné aktualizaci. Všechny ostatní návratové hodnoty způsobí vrácení informace o neúspěšné aktualizaci.

```
1  Function Update(Klíč, PůvodníData, NováData)
       Result: Výsledek aktualizace (Ano, Ne)
       // Dopředný průchod stromem
2      AktuálníUzel ← NačístUzel(ČísloKořenovéhoUzlu);
3      while AktuálníUzel je vnitřní uzel do
4          ČísloUzlu ← NajítPrvekAktuálníhoUzlu(Klíč);
5          AktuálníUzel ← NačístUzel(ČísloUzlu);
6      end while
7      AktualizovatDataAktuálníhoUzlu(Klíč, PůvodníData, NováData,
       PovolenyDuplicitníKlíče);
8      if Aktualizace proběhla úspěšně then
9          Vrátit informaci o úspěšné aktualizaci;
10     else if Aktualizovaný prvek neexistuje then
11         Vrátit informaci o neúspěšné aktualizaci;
12     end if
```

Algorithm 13: Funkce cCommonBpTree::Update

6 Implementace transakčního zpracování

Další částí této práce je rozšíření rámce o podporu transakčního zpracování za účelem zotavení z chyb.

Jak víme z kapitoly 3, tak pro transakční zpracování je nezbytný, tzv. správce transakcí a log soubor. Ovšem ve verzi rámce, která byla pro tuto práci použita, není s transakcemi ani s log souborem počítáno. Bylo proto nutné nejdříve navrhnout strukturu tříd, které budou z transakcemi a log souborem pracovat.

V dalším kroku bylo potřeba jednotlivé třídy a jejich funkce implementovat a následně pak zahrnout při zpracování jednotlivých operací (vkládání, mazání, aktualizace) nad DS (B+strom a sekvenční pole).

6.1 Návrh tříd

Jak je z obrázku 12 patrné, tak pro práci s transakcemi byly vytvořeny 3 základní třídy.

- **cTransactionManagement**

  Třída je určena pro správu transakcí a implementuje operace BEGIN TRANSACTION, COMMIT a ROLLBACK. Reprezentuje tedy správce transakcí (viz kapitola 3)

- **cTransaction**

  Třída reprezentuje transakci jako takovou. Její instance má jedinečný identifikátor a pole změněných stránek (uzlů) DS (pole *mModifiedPages* z obrázku 12).

- **cLog**

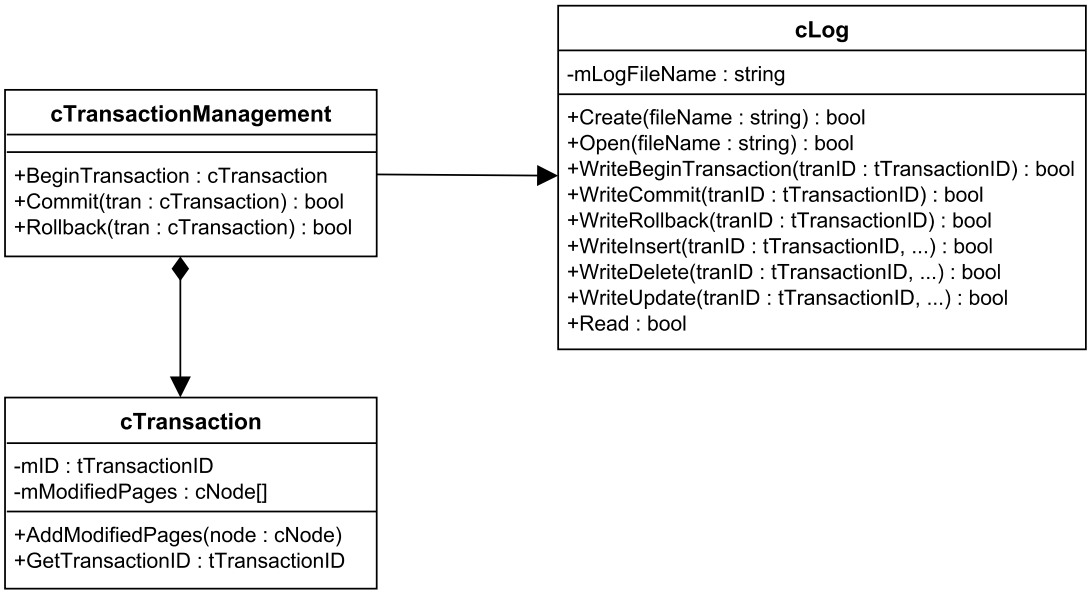
  Slouží pro práci s log souborem. Umožňuje zápis jednotlivých operací do log souboru a zároveň umožňuje jeho čtení.

Struktura výše uvedených tříd byla do rámce zahrnuta přes třídu *cQuickDB*, jak můžeme vidět na obrázku 13. Při vytváření instance třídy *cQuickDB* se rovněž vytvoří instance třídy *cTransactionManagement* a třídy *cLog*. Po vytvoření instance třídy *cLog* se také vytvoří fyzický soubor na disku, podobně jako u třídy *cNodeCache*. Důsledkem toho je, že při vytváření instance třídy *cQuickDB* je potřeba uvést nejenom cestu k datovému souboru ale také cestu k log souboru.

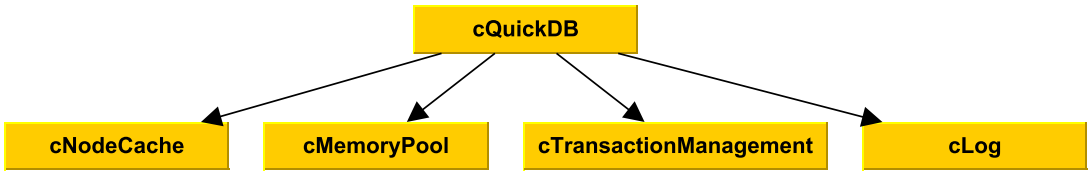
6.2 Log soubor

Jak již bylo zmíněno, tak log soubor je reprezentován třídou *cLog*. Ta pro každou operaci (mazání, vkládání, COMMIT, ...) implementuje funkci, která se stará o uložení všech potřebných informací do log souboru. Všechny informace jsou pak ukládány ve formě binárních dat.

Každá operace je do log souboru uložena jako jeden záznam. Každý záznam je pak identifikován, tzv. typem záznamu, což je jedno bajtové číslo, které je ve třídě *cLog* reprezentováno konstantou (viz níže). Každý záznam může mít různý počet informací a délku v závislosti na typu záznamu.



Obrázek 12: Třídní diagram transakčního zpracování



Obrázek 13: Třída cQuickDB s transakcemi

- `RECORD_TYPE_BEGIN` - Označuje záznam operace BEGIN TRANSACTION.
- `RECORD_TYPE_COMMIT` - Označuje záznam operace COMMIT.
- `RECORD_TYPE_ROLLBACK` - Označuje záznam operace ROLLBACK.
- `RECORD_TYPE_INSERT` - Označuje záznam operace vkládání.
- `RECORD_TYPE_DELETE` - Označuje záznam operace mazání.
- `RECORD_TYPE_UPDATE` - Označuje záznam operace aktualizace.

Obecná struktura záznamu v log souboru je patrná z tabulky 1. V tabulce můžeme vidět, že např. pro uložení operace COMMIT nám stačí 5 B, resp. typ záznamu a ID transakce. Ovšem pro uložení operace vkládání je potřeba zaznamenat mnohem více informací, jako je název DS (jednoznačný identifikátor), vkládaný klíč, atd. Dále je z tabulky 1 patrné, že struktura záznamu je vhodná pro použití operace REDO i UNDO (viz kapitola 3.2).

1B	4B	64B	4B	?	4B	?	4B	?
Typ záznamu	**ID transakce**	**Název DS**	**Velikost klíče**	**Klíč**	**Velikost dat**	**Data**	**Velikost původních dat**	**Původní data**
BEGIN TRANSACTION COMMIT ROLLBACK								
INSERT DELETE								
UPDATE								

Tabulka 1: Struktura záznamu v log souboru

6.3 Operace BEGIN TRANSACTION

Operace BEGIN TRANSACTION je reprezentována funkcí *BeginTransaction* ve třídě *cTransactionManagement*.

Popis algoritmu

- Jak můžeme vidět z algoritmu 14, tak v prvním kroku je získána instance třídy *cTransaction*. Instance je vzata z před alokovaného pole, které je součástí třídy *cTransactionManagement*. V případě, že je toto pole vyčerpáno, dojde k jeho realokaci.
- V dalším kroku uložíme informaci o započaté transakci do log souboru.

6.4 Operace COMMIT

Operace COMMIT je reprezentována funkcí *Commit* rovněž ve třídě *cTransactionManagement*.

```
1 Function BeginTransaction()
      Result: Instance transakce
2     Transakce ← Získat instanci transakce z předalokovaného pole;
3     IDTransakce ← ID transakce Transakce;
4     ZapsatDoLogSouboru(IDTransakce);
```

Algorithm 14: Funkce cTransactionManagement::BeginTransaction

Popis algoritmu

- Z pravidla dopředného zápisu (viz kapitola 3) lze předvídat, že v prvním kroku bude potřeba zapsat informaci do log souboru (viz řádky 2 - 3 algoritmu 15).
- Po zapsaní informace do log souboru projdeme všechny uzly, které byli v průběhu celé transakce změněny (ukazatele na změněné uzly jsou uloženy ve třídě *cTransaction*, jejíž instance je předávána jako parametr této funkce). Následně jsou tyto uzly fyzicky uloženy do datového souboru na disku (viz řádky 4 - 7 algoritmu 15). Z již zmíněných informací víme, že pro práci s datovým souborem je určena třída *cNodeCache*. Proto nám bude zajišťovat i zápis do datového souboru.
- Poté provedeme inicializaci počtu změněných uzlů (viz 8. řádek algoritmu 15).

```
1 Function Commit(Transakce)
      Result: Výsledek operace (Ano, Ne)
2     IDTransakce ← ID transakce Transakce;
3     ZapsatDoLogSouboru(IDTransakce);
4     for i = 0 to Počet změněných uzlů do
5         Uzel ← Změněný uzel i;
6         ZapsatDoDatovéhoSouboru(Uzel);
7     end for
8     Inicializovat počet změněných uzlů;
```

Algorithm 15: Funkce cTransactionManagement::Commit

6.5 Operace ROLLBACK

Operace ROLLBACK je reprezentována funkcí *Rollback* ve třídě *cTransactionManagement*. Aktuální implementace obsahuje pouze zápis do log souboru, tzn. že nedochází ke skutečné realizaci operace UNDO.

6.6 Transakce pro stránkované DS

Doposud jsme si popsali třídy a funkce pro práci s transakcemi, které tvoří jakési jádro transakčního zpracování. Nyní se podíváme, jakým způsobem je možné transakční zpracování povolit a jaké typy transakčního zpracování jsou v implementaci rozlišovány.

Aby bylo možné transakční zpracování zakázat, resp. povolit globálně pro všechny stránkované DS, byla pro tento účel vytvořená proměnná ve třídě *cDStructHeader*. Tato proměnná je pak dostupná přes veřejnou metodu. Toto řešení je obdobou např. povolení, resp. zakázání duplicitních hodnot klíče. Při vytváření DS je pak možné zvolit zda chceme aby bylo transakční zpracování povoleno či nikoli.

Je zřejmé, že pro zavedení transakčního zpracování do DS je potřeba upravit také funkce reprezentující jednotlivé operace (mazání, vkládání a aktualizace). Konkrétní úpravy těchto funkcí si popíšeme v následujících kapitolách. Obecně však můžeme říci, že všechny funkce byli rozšířeny o nepovinný parametr typu *cTransaction*, který jak dobře víme reprezentuje danou transakci. Vzhledem k tomu, že je tento parametr nepovinný, rozlišujeme dva způsoby volání operací, tedy bez transakce a s transakcí. Podle způsobu volání konkrétní operace pak rozlišujeme dva typy transakčního zpracování:

1. **Single Transaction (samostatná transakce)**

   Každá operace je prováděna v rámci samostatné transakce. Tento typ zpracování nastane v případě, kdy při volání operace není zadána konkrétní transakce v rámci, které by se měla operace provést. Na začátku každé operace je proto provedena operace BEGIN TRANSACTION. Po úspěšném provedení každé operace se pak provádí operace COMMIT.

2. **Bulk Transaction (hromadná transakce)**

   Všechny operace jsou prováděny v rámci jedné transakce. Tento typ zpracování nastane v případě, kdy při volání operace zadáme konkrétní transakci v rámci, které by se měla operace provést. Operace COMMIT se provádí explicitně po provedení všech požadovaných operací.

6.7 Transakce v B+stromu

Nyní si ukážeme úpravy implementace B+stromu způsobené zavedením transakčního zpracování. Veškeré úpravy byly v tomto případě provedeny v rámci třídy *cCommonBpTree* a *cPagedTree*.

Z předchozí kapitoly víme, že funkce reprezentující jednotlivé operace byly rozšířeny o nepovinný parametr a vysvětlili jsme si rozdíl mezi Single Transaction a Bulk Transaction. Dále si tedy ukážeme, jak jsou jednotlivé typy transakcí implementovány a jak se během provádění operace dále s transakcí pracuje.

Všechny úpravy způsobené zavedením transakčního zpracování si ukážeme na operaci aktualizace (viz algoritmus 16). U ostatních operací jsou úpravy v principu stejné a není potřeba je všechny uvádět.

Pokud porovnáme původní algoritmus aktualizace 13 s rozšířeným algoritmem 16, tak můžeme vidět že přibyli řádky 2 - 4, které nám zajišťují vytvoření Single Transaction v případě, že není zadána konkrétní transakce.

Po dalším zkoumání algoritmu 16 zjistíme, že přibyl také řádek 12, který nám zajišťuje uložení změněného uzlu do transakce. Z předchozí kapitoly víme, že uložení změněných uzlů je nezbytné k provedení operace COMMIT, resp. ROLLBACK.

Pokud je operace prováděná v rámci Single Transaction, pak víme že je potřeba po každé operaci provést operaci COMMIT, což je patrné z řádků 17 - 19 v algoritmu 16.

```
1  Function Update(Klíč, PůvodníData, NováData, Transakce)
       Result: Výsledek aktualizace (Ano, Ne)
2      if Transakce je prázdná then
3          Transakce ← Vytvořit novou transakci;
4      end if
       // Dopředný průchod stromem
5      AktuálníUzel ← Kořenový uzel;
6      while AktuálníUzel je vnitřní uzel do
7          AktuálníUzel ← Načti podřízený uzel uzlu AktuálníUzel pro klíč Klíč;
8      end while
9      AktualizovatDataAktuálníhoUzlu(Klíč, PůvodníData, NováData,
       PovolenyDuplicitníKlíče);
10     if Aktualizace proběhla úspěšně then
11         UložitZměněnýUzel(AktuálníUzel);
12         Vrátit informaci o úspěšné aktualizaci;
13     else if Aktualizovaný prvek neexistuje then
14         Vrátit informaci o neúspěšné aktualizaci;
15     end if
16     if Transakce je samostatná then
17         COMMIT(Transakce);
18     end if
```

Algorithm 16: Funkce cCommonBpTree::Update s transakcemi

6.8 Transakce v sekvenčním poli

Součástí implementace je také zavedení transakčního zpracování do sekvenčního pole. Proto si nyní ukážeme jaké úpravy bylo potřeba pro tuto DS provést. Veškeré úpravy byly v tomto případě provedeny ve třídě *cSequentialArray*.

Opět si úpravy při zavedení transakčního zpracování do sekvenčního pole demonstrujeme pouze na jedné operaci a to operaci vkládání. Pro zbylé operace jsou úpravy v principu stejné a proto není potřeba je všechny uvádět.

Z algoritmu 17 je patrné, že úpravy jsou podobné úpravám v B+stromu. Liší se pouze v umístění funkce pro uložení změněného uzlu, které závislé na implementaci konkrétní operace (viz řádky 8 a 16 algoritmu 17). Vytvoření samostatné transakce je opět na začátku celé funkce, což je možné vidět na řádcích 2 - 4 algoritmu 17. Stejně, tak je na konci funkce volána operace COMMIT (viz řádky 18 - 20 algoritmu 17).

```
1  Function AddItem(Prvek, Transakce)
       Result: Výsledek vkládání (Ano, Ne)
2      if Transakce je prázdná then
3          Transakce ← Vytvořit novou transakci;
4      end if
5      AktuálníUzel ← Poslední uzel;
6      if AktuálníUzel má dostatek místa then
7          VložPrvekDoAktuálníhoUzlu(Prvek);
8          UložitZměněnýUzel(AktuálníUzel);
9      else
10         if AktuálníUzel má sousední uzel then
11             AktuálníUzel ← Sousední uzel uzlu AktuálníUzel;
12         else
13             AktuálníUzel ← Vytvořit nový uzel;
14         end if
15         VložPrvekDoAktuálníhoUzlu(Prvek);
16         UložitZměněnýUzel(AktuálníUzel);
17     end if
18     if Transakce je samostatná then
19         COMMIT(Transakce);
20     end if
21     Vrátit informaci o úspěšné aktualizaci;
```

Algorithm 17: Funkce cSequentialArray::AddItem s transakcemi

7 Experimenty

7.1 Cíle

Experimenty, které byly v rámci této práce provedeny, si kladou za cíl změřit vliv implementace transakčního zpracování na výkonnost prováděných operací mazání, vkládání a aktualizace. Cíle můžeme rozdělit do dvou bodů:

- Změřit výkonnost operací B+stromu se zapnutým transakčním zpracováním a bez něj.
- Změřit výkonnost operace vkládání sekvenčního pole se zapnutým transakčním zpracováním a bez něj.

7.2 Postup měření

1. Zvolit režim testování (bez transakcí, s hromadnými transakcemi, se samostatnými transakcemi).
2. Zvolit počet testovaných dat.
3. Provést operaci vkládání pro celou množinu testovaných dat.
4. Změřit dobu zpracování, počet operací za sekundu, popř. dobu zpracování operace *COMMIT*.
5. Provést operaci aktualizace pro celou množinu testovaných dat.
6. Změřit hodnoty z bodu 4.
7. Provést operaci mazání pro celou množinu testovaných dat.
8. Změřit hodnoty z bodu 4.

7.3 Testovací prostředí

Všechna měření byla prováděna na notebooku značky Dell řady Latitude E6540. Konfiguraci můžeme vidět v tabulce 2.

7.4 Testovací aplikace

Pro potřeby měření byly vytvořeny dvě testovací aplikace. Jedna vytváří B+strom a umožňuje testovat operace mazání, vkládání a aktualizace. Druhá pak vytváří sekvenční pole a umožňuje testovat operaci vkládání, která je jako jediná v této DS implementována. Obě aplikace měří dobu zpracování prováděných operací, počet provedených operací za sekundu a v případě hromadné transakce je měřena doba zpracování operace *COMMIT*.

Parametry pro testování jsou následující:

Procesor:	Intel(R) Code(TM) i5-4300M CPU 2.60 GHz
Paměť (RAM):	8,00 GB
Platforma:	x64
Pevný disk:	SATA 500 GB (7200 ot./min.)
Operační systém:	Windows 8.1 Enterprise (64-bit)

Tabulka 2: Konfigurace testovacího počítače

- CACHE SIZE - Počet uzlů, jenž budou uloženy v paměti.
- BLOCK SIZE - Velikost bloku (stránky) na disku (násobek 512 bajtů).
- INMEM SIZE - Velikost bloku v hlavní paměti (> BLOCK SIZE).
- DSMODE - Režim DS. Může nabývat několika hodnot a umožňuje např. povolení komprese, atd.
- DSCODE - Kód DS[3]. Nabývá dvou hodnot a umožňuje povolení duplicitních klíčů.
- ITEM TYPE - Typ klíče.
- TUPLE LENGTH - Velikost vektoru klíče.
- DATA LENGTH - Velikost dat.
- ITEM COUNT - Množství dat pro testování operací.

7.5 Měření B+stromu

Měření výkonnosti operací prováděných v B+stromu bylo spuštěno s parametry testovací aplikace podle tabulky 7.

CACHE SIZE	**10000**
BLOCK SIZE	**8192**
MAX NODE INMEM SIZE	**10240**
DSMODE	**DEFAULT**
DSCODE	**BTREE**
ITEM TYPE	**TUPLE**
TUPLE LENGTH	**5**
DATA LENGTH	**4**

Tabulka 3: Nastavení parametru při testování B+stromu

Po nastavení parametrů byla provedena měření ve třech různých režimech. V každém z těchto režimů pak proběhla dílčí měření s odlišným množstvím dat pro všechny operace.

[3]Specifický parametr B+stromu.

Poznámka 7.1 Všechny výsledky měření je možné vidět na obrázcích v příloze A, na které bude dále v textu odkazováno.

1. **Režim bez transakčního zpracování**

   V tomto režimu nebylo povoleno transakční zpracování. Nejprve byla testována operace vkládání, poté aktualizace a nakonec mazání. Všechny operace proběhly vždy pro celou množinu dat.

ITEM SIZE	Operace	Doba zpracování [s]	Výkonnost [op/s]
100 000	INSERT	0,437	229 876,4
100 000	UPDATE	0,320	312 500,1
100 000	DELETE	1,169	85 543,2
1 000 000	INSERT	4,928	203 769,3
1 000 000	UPDATE	4,019	248 818,1
1 000 000	DELETE	12,621	79 233,0
2 000 000	INSERT	10,457	192 057,0
2 000 000	UPDATE	9,715	205 867,2
2 000 000	DELETE	31,160	67 184,9

Tabulka 4: B+strom - výsledky měření bez transakčního zpracování

   Dílčí výsledky jednotlivých měření můžeme vidět v tabulce 4. Porovnání závislosti množství dat na výslednou dobu zpracování a výkonnosti můžeme vidět na obrázku 14 a 15.

   Z výsledků je patrné, že nejrychlejší a nejvýkonnější operací je v tomto případě operace *UPDATE*, tedy aktualizace. Vzhledem k jednoduchosti této operace se nejedná o žádné překvapivé zjištění.

   Nejhoršího výsledku dosáhla operace *DELETE*, tedy mazání. V některých případech se jedná až o 3x delší dobu zpracování v porovnání se zbylými operacemi. Jelikož operace mazání je v principu složitější než zbylé dvě operace, tak určitý pokles výkonu lze předpokládat.

2. **Režim hromadné transakce**

   V tomto režimu již bylo transakční zpracování povoleno. Pořadí prováděných operací a dílčí měření zůstaly stejné jako v předchozím režimu. Navíc pak byli vytvořeny 3 transakce, každá pro jeden typ operace. Postup měření je zřejmý z algoritmu 18, kde můžeme vidět příklad měření operace vkládání. Stejný postup měření je rovněž použit i u zbylých dvou operací.

   Dílčí výsledky jednotlivých měření můžeme vidět v tabulce 5. Grafické znázornění těchto výsledků je pak znázorněno na obrázcích 16, 17 a 18.

   Jak si je možné všimnout (viz tabulka 5), tak oproti předchozímu měření byl naměřen další údaj o době zpracování operace *COMMIT*, která byla provedena po jednotlivých operacích (viz obrázek 17). Z výsledků je patrné, že operace *COMMIT*

```
1 Transakce ← BeginTransaction;
2 for i = 1 to ITEMCOUNT do
3 |   Insert(Klíč, Data, Transakce);
4 end for
5 Commit(Transakce);
```

Algorithm 18: Měření v režimu hromadné transakce

ITEM SIZE	Operace	Doba zpracování [s]	Doba zpracování op. COMMIT [s]	Výkonnost [op/s]
100 000	INSERT	1,730	13,592	58 067,1
100 000	UPDATE	2,096	7,639	47 709,9
100 000	DELETE	2,610	7,812	38 314,2
1 000 000	INSERT	33,978	129,119	29 553,7
1 000 000	UPDATE	44,460	74,787	22 492,0
1 000 000	DELETE	50,295	76,106	19 882,7
2 000 000	INSERT	85,660	194,080	23 445,5
2 000 000	UPDATE	159,092	151,408	12 571,3
2 000 000	DELETE	168,444	155,758	11 873,4

Tabulka 5: B+strom - výsledky měření s hromadnou transakcí

je obecně nejnákladnější než samotná operace vkládání, aktualizace nebo mazání. Je to zřejmě způsobeno nákladným ukládáním změněných uzlů do datového souboru na disku. Celková doba zpracování operace je tedy dána součtem doby zpracování operace a doby zpracování operace COMMIT.

Z výsledků je dále patrné, že doba zpracování jednotlivých operací se značně prodloužila, což můžeme vidět na obrázku 19. Tento nárůst je pochopitelný z důvodu dopředného zápisu do log souboru, který je při zapnutém transakčním zpracování prováděn. Největší nárůst můžeme zaznamenat u operace *UPDATE* při zpracování 2 000 000 prvků, kde došlo ke zvýšení doby z 9,175[s] na 159,092[s], což je velice znepokojivé zjištění.

Nárůst jsme zaznamenali také u provedených operací za sekundu, což je patrné z obrázku 20.

3. **Režim samostatné transakce**

   V tomto režimu bylo opět transakční zpracování povoleno. Operace se však neprováděli v rámci explicitně definované transakce jako v předchozím měření, ale každá operace byla provedena v samostatné transakci. O vytvoření samostatné transakce se již stará každá operace sama.

   Vzhledem k časové náročnosti tohoto měření bylo provedeno jedno dílčí měření pro 10 000 prvků.

ITEM SIZE	Operace	Doba zpracování [s]	Výkonnost [op/s]
10 000	INSERT	134,842	74,5
10 000	UPDATE	117,393	84,9
10 000	DELETE	119,393	83,8

Tabulka 6: B+strom - výsledky měření se samostatnou transakcí

Výsledky měření můžeme vidět v tabulce 6. Z výsledků je patrné, že provádění operací v rámci samostatných transakcí je výkonnostní problém. Důvodem je provedení operace *COMMIT* po každé operaci vkládání, aktualizace nebo mazání.

7.6 Měření sekvenčního pole

Vzhledem k chybějící implementaci operace mazání a aktualizace byla měření prováděna pouze pro operaci vkládání. Měření bylo spuštěno s parametry testovací aplikace podle tabulky 7.

CACHE SIZE	**1000**
BLOCK SIZE	**8192**
MAX NODE INMEM SIZE	**10240**
DSMODE	**DEFAULT**
ITEM TYPE	**TUPLE**
TUPLE LENGTH	**5**

Tabulka 7: Nastavení parametru při testování sekvenčního pole

Stejně jako měření B+stromu, tak i toto měření bylo provedeno ve třech režimech. Stejně tak postup měření byl totožný, jako v případě B+stromu a dále již v textu nebude zmiňován.

1. **Režim bez transakčního zpracování**

   Výsledky měření můžeme vidět v tabulce 8.

ITEM SIZE	Operace	Doba zpracování [s]	Výkonnost [op/s]
100 000	INSERT	0,063	1 587 302,5
1 000 000	INSERT	58,427	17 115,4
2 000 000	INSERT	155,311	12 877,4

Tabulka 8: Sekvenční pole - výsledky měření bez transakčního zpracování

2. **Režim hromadné transakce**

   Výsledky měření můžeme vidět v tabulce 9. Změna doby zpracování operace *COMMIT* v závislosti na množství dat je patrná na obrázku 22.

ITEM SIZE	Operace	Doba zpracování [s]	Doba zpracování op. COMMIT [s]	Výkonnost [op/s]
100 000	INSERT	1,234	8,647	81 037,3
1 000 000	INSERT	94,797	17,082	10 548,9
2 000 000	INSERT	258,412	19,163	7 739,6

Tabulka 9: Sekvenční pole - výsledky měření s hromadnou transakcí

3. **Režim samostatné transakce**

   V tomto režimu bylo stejně jako u B+stromu provedeno pouze jedno dílčí měření pro 10 000 prvků. Výsledek je možno vidět v tabulce 10.

ITEM SIZE	Operace	Doba zpracování [s]	Výkonnost [op/s]
10 000	INSERT	183,059	54,6

Tabulka 10: Sekvenční pole - výsledky měření se samostatnou transakcí

Porovnání výsledků měření v jednotlivých režimech můžeme vidět na obrázku 21 a 23.

7.7 Vyhodnocení

Z výsledků měření je zřejmé, že implementace transakčního zpracování má nemalý vliv na výkonnost prováděných operací a to jak v B+stromu, tak v sekvenčním poli.

Největší pokles výkonnosti jsme mohli zaznamenat u operace aktualizace, kde byl pokles o více jak 95%. Toto zjištění je velice překvapivé, protože tato operace by měla být řádově rychlejší než operace vkládání a mazání. V případě dalšího rozšíření této práce by bylo vhodné tuto operaci optimalizovat.

8 Závěr

8.1 Zhodnocení

Hlavním cílem této práce bylo porovnat vlastnosti implementace sekvenčního pole a B+stromu. Bohužel se tento cíl podařilo splnit jen částečně. Operace, které měly být implementovány u obou těchto DS byli nakonec implementovány pouze u B+stromu. Důvodem byly mimo jiné problémy vzniklé při implementaci operace mazání (viz kapitola 5.1.9).

Dalším cílem této práce bylo rozšíření rámce RadegastDB o transakční zpracování, tedy zápis do log souboru a zotavení z chyb. Tento cíl se podařilo rovněž splnit jen částečně. Byl implementován zápis do log souboru ovšem bez možnosti zotavení.

Hlavní přínos této práce je viditelný v rozšíření rámce RadegastDB o operace mazání a aktualizace B+stromu, které rámec do té doby neimplementoval. Při implementaci bylo vyřešeno netriviální množství problémů a dá se předpokládat, že toto řešení se stane základem k dalšímu vývoji. Díky této práci byl také vytvořen základ pro práci z transakcemi a log souborem.

Dalším nemalým přínosem jsou také výsledky měření, které odhalily vliv zápisu do log souboru na celkový výkon prováděných operací. Tyto výsledky mohou posloužit jako podklady k dalšímu vývoji a optimalizaci jednotlivých operací.

8.2 Další vývoj projektu

Jedním z námětů k dalšímu vývoji projektu je jistě rozšíření operace mazání u B+stromu o podporu variabilní délky klíče a dat. Aktuální implementace mazání navíc neodpovídá specifikaci B+stromu, jelikož neumožňuje slučování uzlů a přesun mezi uzly, které mají rozdílného předka (viz kapitola 5.1.9). Dalším námětem k rozšíření je tedy implementace mazání podle specifikace B+stromu. Také implementované transakční zpracování lze dále rozšiřovat a optimalizovat, např. o zotavení transakce (UNDO), zotavení odloženou aktualizací (NO-UNDO/REDO), zotavení okamžitou aktualizací (UNDO/REDO), kontrolní body, atd. Právě kontrolní body jsou jistě dobrým námětem k optimalizaci operace COMMIT, kde měření odhalila velký nedostatek.

Dále by bylo užitečné v rámci celého rámce rozšířit i ostatní DS o operaci mazání a aktualizace. Také transakční zpracování je možné zahrnout do ostatních DS.

9 Reference

[1] R. Bayer, E. McCreight. *Organization and Maintenance of Large Ordered Indices*, Acta Informatica, 1972.

[2] Jiří Dvorský. *Algoritmy 1*, VŠB - Technická univerzita Ostrava, 2007.

[3] R. Elmasri, S. B. Navathe. *Fundamentals of Database Systems, 6th Edition*, Addison Wesley, 2010

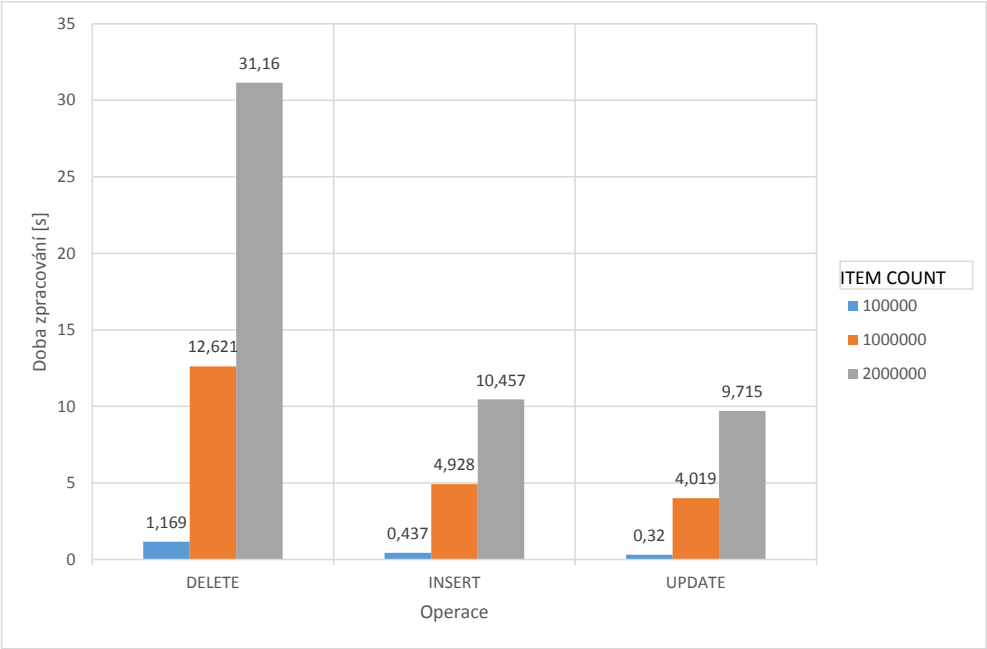
[4] M. Krátký, R. Bača. *Databázové systémy*, VŠB - Technická univerzita Ostrava, 2015.

[5] J. Pokorný. *Databázová abeceda*, Veletiny: SCIENCE, 1998.

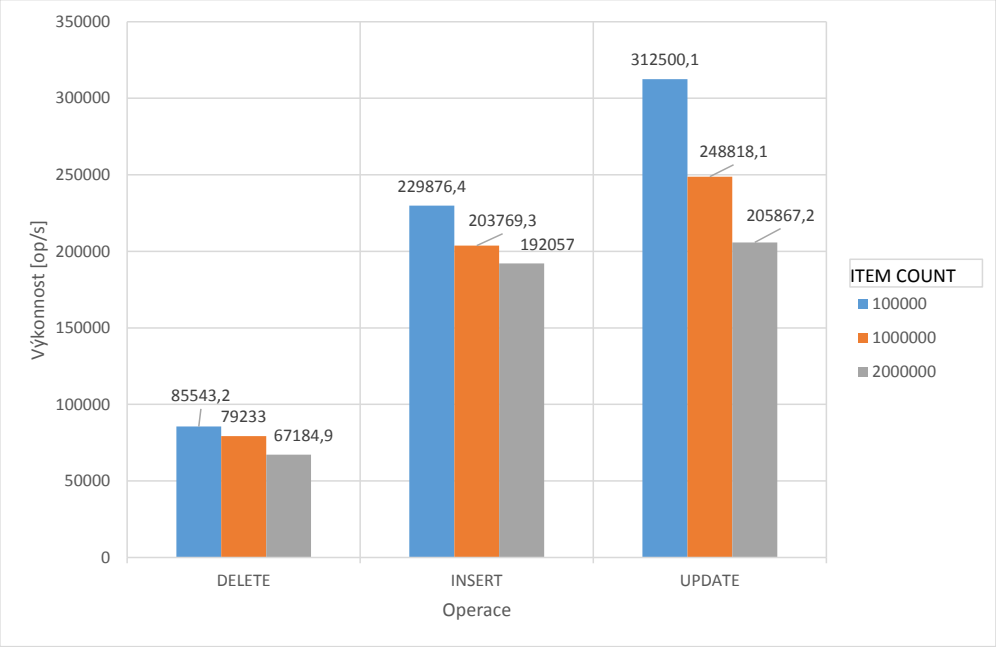
[6] Piotr Wróblewski. *Algoritmy: datové struktury a programovací techniky*, Vyd. 1. Brno: Computer Press, 2004. ISBN 80-251-0343-9

[7] J. Zendulka. Sylaby předmětu Databázové systémy a návrh databází, `http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/9_trans.pdf`, 2004

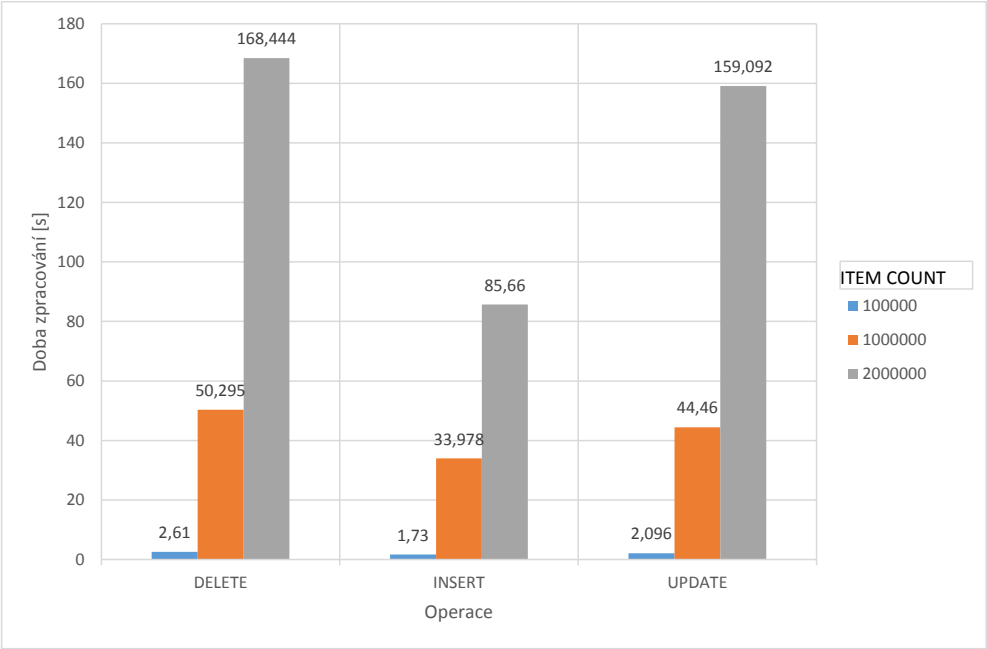
A Grafy



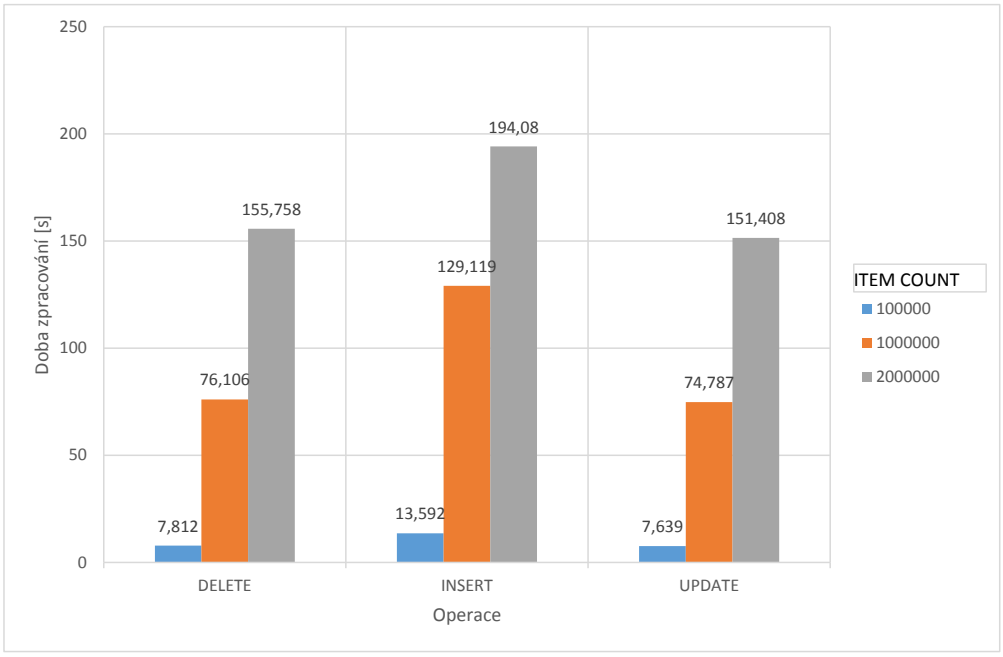
Obrázek 14: B+strom - měření doby zpracování bez transakčního zpracování



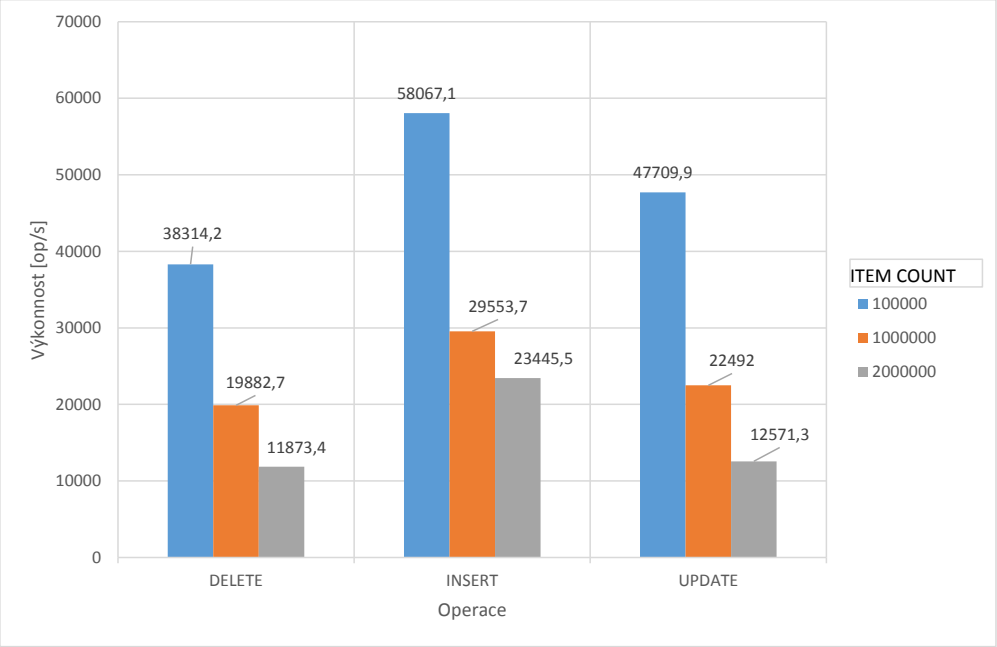
Obrázek 15: B+strom - měření výkonnosti bez transakčního zpracování



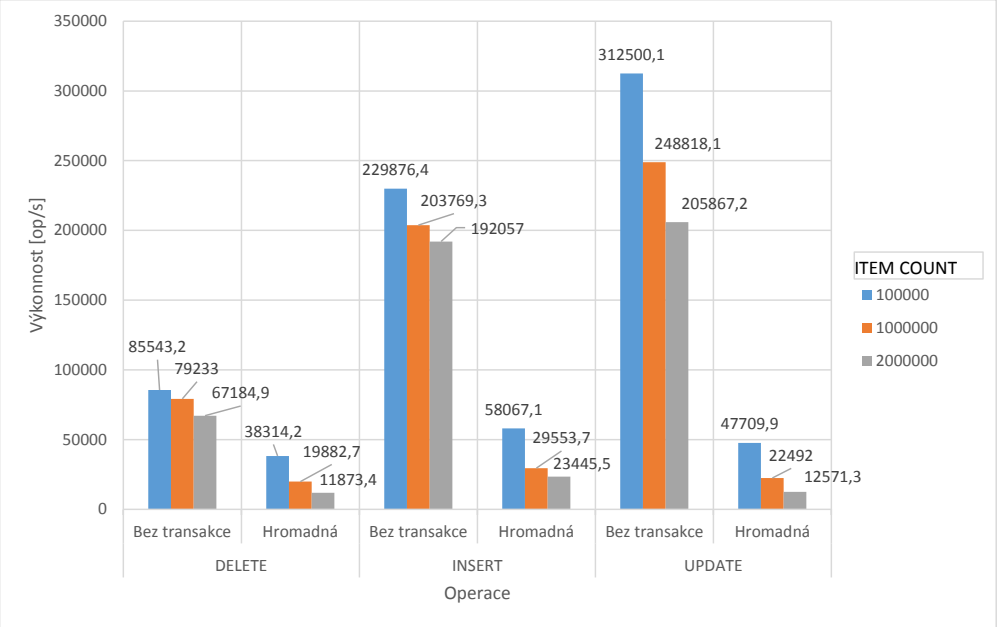
Obrázek 16: B+strom - měření doby zpracování při hromadné transakci



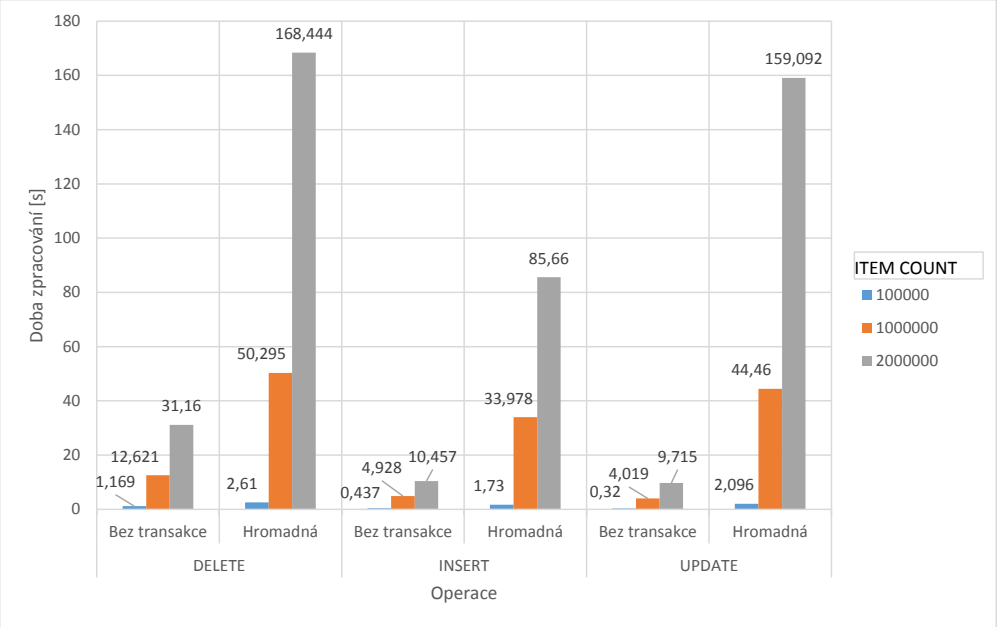
Obrázek 17: B+strom - měření doby zpracování operace COMMIT při hromadné transakci



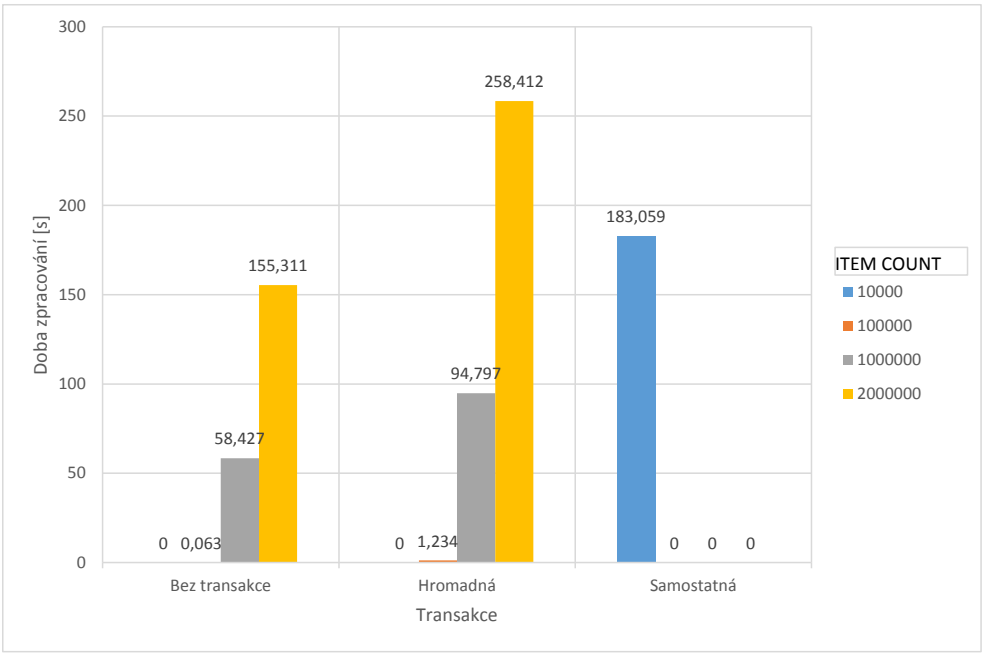
Obrázek 18: B+strom - měření výkonnosti při hromadné transakci



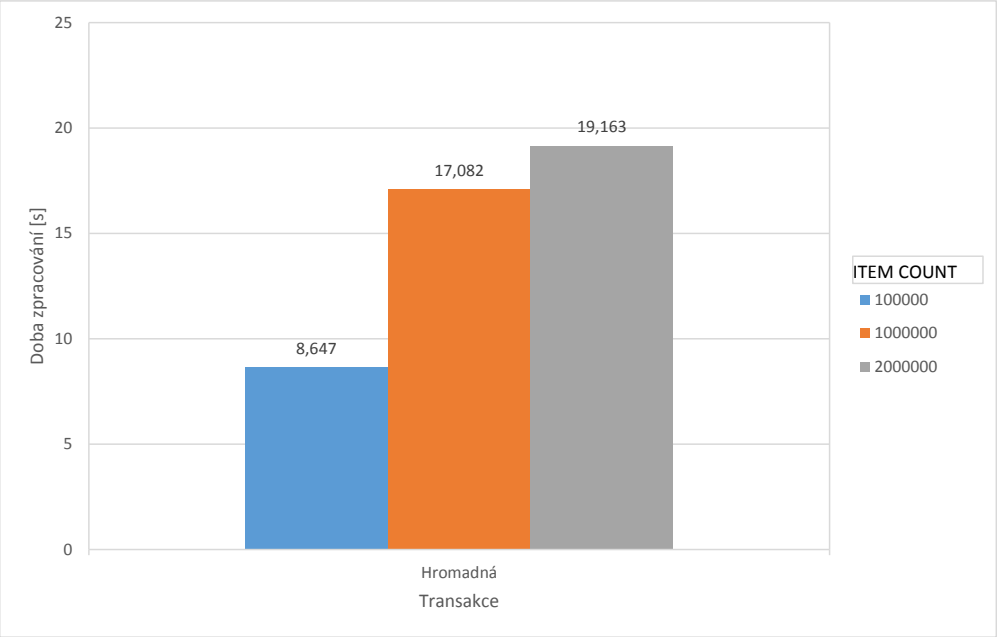
Obrázek 19: B+strom - porovnání doby zpracování v režimu hromadné transakce a v režimu bez transakce



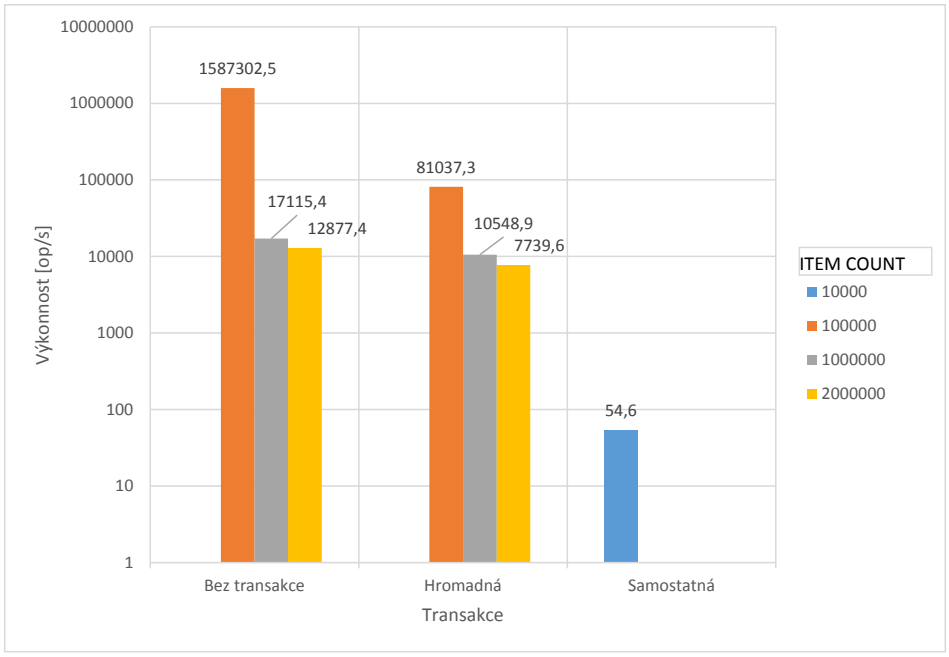
Obrázek 20: B+strom - porovnání výkonnosti v režimu hromadné transakce a v režimu bez transakce



Obrázek 21: Sekvenční pole - měření doby zpracování operace vkládání



Obrázek 22: Sekvenční pole - měření doby zpracování operace COMMIT po operaci vkládání



Obrázek 23: Sekvenční pole - měření výkonnosti operace vkládání

B Příloha na CD

\src	Zdrojové kódy rámce RadegastDB a testovací aplikace
\src\common	Zdrojové kódy podpůrných tříd
\src\dstruct	Zdrojové kódy tříd tvořící jádro datových struktur
\src\test	Testovací aplikace
\src\test\paged\btree_test	Testovací aplikace B+stromu
\src\test\paged\sequentialarray_test	Testovací aplikace sekvenčního pole
\dp	Diplomová práce a zadání v elektronické podobě

Tabulka 11: Obsah CD